|  |
| --- |
| Java知识 |
| Java后台开发 |
| 本文档整理了java面试相关的知识点，以便自己在平时查漏补缺。同时方便以后再次面试时，可以随时查阅复习。 |

2018-10-14

[面试知识 0](#_Toc527321199)

[计算机基础 8](#_Toc527321200)

[十进制转二进制 8](#_Toc527321201)

[整数 8](#_Toc527321202)

[小数 8](#_Toc527321203)

[原码、反码、补码 8](#_Toc527321204)

[原码 8](#_Toc527321205)

[反码 9](#_Toc527321206)

[补码 9](#_Toc527321207)

[浮点数的二进制表示方法（IEEE754标准） 9](#_Toc527321208)

[补码运算 10](#_Toc527321209)

[加法运算 10](#_Toc527321210)

[减法运算 10](#_Toc527321211)

[乘法运算 11](#_Toc527321212)

[除法运算 11](#_Toc527321213)

[Java基础知识 12](#_Toc527321214)

[基本类型与对应封装类型 12](#_Toc527321215)

[基本类型 12](#_Toc527321216)

[封装类型 12](#_Toc527321217)

[装箱 12](#_Toc527321218)

[拆箱 12](#_Toc527321219)

[引用类型 12](#_Toc527321220)

[强引用 12](#_Toc527321221)

[软引用 12](#_Toc527321222)

[弱引用 12](#_Toc527321223)

[虚引用 12](#_Toc527321224)

[运算符 12](#_Toc527321225)

[算术运算符 12](#_Toc527321226)

[关系运算符 12](#_Toc527321227)

[位运算符 12](#_Toc527321228)

[逻辑运算符 13](#_Toc527321229)

[赋值运算符 13](#_Toc527321230)

[条件运算符 13](#_Toc527321231)

[Instanceof运算符 13](#_Toc527321232)

[数组 13](#_Toc527321233)

[关键字 13](#_Toc527321234)

[参数传递 13](#_Toc527321235)

[值传递 13](#_Toc527321236)

[引用传递 13](#_Toc527321237)

[IO流 13](#_Toc527321238)

[输出流、输入流 13](#_Toc527321239)

[Java IO 13](#_Toc527321240)

[Java NIO 13](#_Toc527321241)

[Java 8 Stream 13](#_Toc527321242)

[Java序列化 13](#_Toc527321243)

[多线程 13](#_Toc527321244)

[多进程 14](#_Toc527321245)

[多线程 14](#_Toc527321246)

[线程池 14](#_Toc527321247)

[对象池 14](#_Toc527321248)

[Java.util.Collection类 14](#_Toc527321249)

[List 14](#_Toc527321250)

[Set 14](#_Toc527321251)

[Queue 15](#_Toc527321252)

[Java.util.Map类型 15](#_Toc527321253)

[HashMap 15](#_Toc527321254)

[HashTable 15](#_Toc527321255)

[SoftHashMap 15](#_Toc527321256)

[WeakHashMap 15](#_Toc527321257)

[SortedMap 15](#_Toc527321258)

[Java.util.Date 15](#_Toc527321259)

[Java.util.concurrent包 15](#_Toc527321260)

[ConcurrentHashMap 15](#_Toc527321261)

[ConcurrentLinkedQueue 15](#_Toc527321262)

[ConcurrentLinkedDeque 16](#_Toc527321263)

[Jdk版本 16](#_Toc527321264)

[Java8 16](#_Toc527321265)

[Lambda 16](#_Toc527321266)

[Stream 16](#_Toc527321267)

[函数式接口 16](#_Toc527321268)

[反射机制 16](#_Toc527321269)

[注解 16](#_Toc527321270)

[Java常见命令 16](#_Toc527321271)

[网络知识 16](#_Toc527321272)

[OSI七层网络模型 16](#_Toc527321273)

[TCP/IP协议 16](#_Toc527321274)

[TCP/UDP 16](#_Toc527321275)

[Http/Https 16](#_Toc527321276)

[RPC协议 16](#_Toc527321277)

[短连接与长连接 17](#_Toc527321278)

[网络安全知识 17](#_Toc527321279)

[签名与验签 17](#_Toc527321280)

[对称加密 17](#_Toc527321281)

[非对称加密 17](#_Toc527321282)

[算法与数据结构 17](#_Toc527321283)

[数据结构 17](#_Toc527321284)

[表（列表） 17](#_Toc527321285)

[栈 17](#_Toc527321286)

[队列 17](#_Toc527321287)

[树 17](#_Toc527321288)

[堆 17](#_Toc527321289)

[图 17](#_Toc527321290)

[算法 18](#_Toc527321291)

[排序 18](#_Toc527321292)

[查找 18](#_Toc527321293)

[设计模式（23种） 18](#_Toc527321294)

[创建型模式 18](#_Toc527321295)

[工厂方法模式 18](#_Toc527321296)

[抽象工厂模式 18](#_Toc527321297)

[单例模式 18](#_Toc527321298)

[创建者模式 18](#_Toc527321299)

[原型模式 19](#_Toc527321300)

[结构型模式 19](#_Toc527321301)

[适配器模式 19](#_Toc527321302)

[装饰器模式 19](#_Toc527321303)

[代理模式 19](#_Toc527321304)

[外观模式 19](#_Toc527321305)

[桥接模式 19](#_Toc527321306)

[组合模式 19](#_Toc527321307)

[亨元模式 19](#_Toc527321308)

[行为型模式 19](#_Toc527321309)

[策略模式 19](#_Toc527321310)

[模板方法模式 19](#_Toc527321311)

[观察者模式 19](#_Toc527321312)

[迭代子模式 19](#_Toc527321313)

[责任链模式 19](#_Toc527321314)

[命令模式 19](#_Toc527321315)

[备忘录模式 20](#_Toc527321316)

[状态模式 20](#_Toc527321317)

[访问者模式 20](#_Toc527321318)

[终结者模式 20](#_Toc527321319)

[解释器模式 20](#_Toc527321320)

[常用框架 20](#_Toc527321321)

[Springboot 20](#_Toc527321322)

[Spring 20](#_Toc527321323)

[Spring mvc 20](#_Toc527321324)

[Mybatis 20](#_Toc527321325)

[Quartz 20](#_Toc527321326)

[Junit 20](#_Toc527321327)

[Dubbo 20](#_Toc527321328)

[FreeMarker 20](#_Toc527321329)

[Zookeeper 20](#_Toc527321330)

[Docket 21](#_Toc527321331)

[Netty 21](#_Toc527321332)

[Mina 21](#_Toc527321333)

[Hadoop 21](#_Toc527321334)

[分布式 21](#_Toc527321335)

[Rpc框架：dubbo 21](#_Toc527321336)

[注册中心：zookeeper 21](#_Toc527321337)

[配置中心：Apollo 21](#_Toc527321338)

[信息队列 21](#_Toc527321339)

[activeMq 21](#_Toc527321340)

[kafka 21](#_Toc527321341)

[rocketMq 21](#_Toc527321342)

[数据存储 21](#_Toc527321343)

[数据库 21](#_Toc527321344)

[Mysql 21](#_Toc527321345)

[Oracle 22](#_Toc527321346)

[缓存 22](#_Toc527321347)

[Memcached 22](#_Toc527321348)

[redis 22](#_Toc527321349)

[codis 22](#_Toc527321350)

[ftp 22](#_Toc527321351)

[ftp 22](#_Toc527321352)

[ftps 22](#_Toc527321353)

[工具和插件 22](#_Toc527321354)

[Eclipse 22](#_Toc527321355)

[Svn 22](#_Toc527321356)

[Git 22](#_Toc527321357)

[Maven 22](#_Toc527321358)

[服务器 22](#_Toc527321359)

[Tomcat 22](#_Toc527321360)

[Jetty 23](#_Toc527321361)

[Nginx 23](#_Toc527321362)

[Linux 23](#_Toc527321363)

[常见命令 23](#_Toc527321364)

[日志查找 23](#_Toc527321365)

[文件查找 23](#_Toc527321366)

[页面前端框架 23](#_Toc527321367)

[Bootstrap 23](#_Toc527321368)

[Jquery 23](#_Toc527321369)

[Highcharts 23](#_Toc527321370)

[Java虚拟机 23](#_Toc527321371)

[面试技巧 23](#_Toc527321372)

# 计算机基础

## 十进制转二进制

### 整数

采用除以2取余的方式

Eg:13=1101[二进制]

13/2=6 ---- 1

6/2=3 ---- 0

3/2=1 ---- 1

1/2=0 ---- 1

所以从下往上去余数为1101

### 小数

采用乘以2取整数部分

Eg：0.5=0.1[二进制]

0.5\*2=1.0 -------- 1

所以1

0.4=0.0110 0110 0110 0110 …

0.4\*2=0.8 -------- 0

0.8\*2=1.6 -------- 1

0.6\*2=1.2 -------- 1

0.2\*2=0.4 -------- 0

…

**所以浮点数在计算机存储可能会出现精度问题。**

## 原码、反码、补码

### 原码

第一位表示符号（0表示正数，1表示负数），后面其余表示值

取值返回-127~127

### 反码

正数的反码是其本身，负数的反码是在**原码**的基础上，符号位不变，其余各位取反

取值返回-127~127

### 补码

正数的补码是其本身，负数的补码是在**反码**的基础上+1

取值返回-128~127，比原码、反码多一位1000 0000表示-128

(-1) + (-127) = [1000 0001]原 + [1111 1111]原 = [1111 1111]补 + [1000 0001]补 = [1000 0000]补

**特殊**：0用原码或者反码都有两种标识，补码只有一种，补码解决了0有两种编码的问题。降级计算机处理复杂性，也就是计算机采用补码作为机器码原因，采用了补码，计算机只有加法运算，且符号位参加运算，降低运算的复杂度

原码：0 = 0000 0000 或者 0 = 1000 0000

反码：0 = 0000 0000 或者 0 = 1111 1111

补码：0 = 0000 0000

|  |  |  |  |
| --- | --- | --- | --- |
| 十进制数 | 原码 | 反码 | 补码 |
| 85 | 0101 0101 | 0101 0101 | 0101 0101 |
| -85 | 1101 0101 | 1010 1010 | 1010 1011 |
| 9 | 0000 1001 | 0000 1001 | 0000 1001 |
| -9 | 1000 1001 | 1111 0110 | 1111 0111 |

## 浮点数的二进制表示方法（IEEE754标准）

现代计算机中，以IEEE 754标准存储浮点数，这个标准在内存中存储的格式：

|  |  |  |
| --- | --- | --- |
| 数符 | 阶码（含阶符） | 尾数 |
| sign | exponent | fraction |

float与double的存储标准

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 | 数符 | 阶码 | 尾数 | 总长度 |
| float | 1 | 8 | 23 | 32 |
| double | 1 | 11 | 52 | 64 |

eg1:

float f = 13.4的二进制

1、数符为0（整数位0，负数为1）

2、整数部分转二进制 13 = 1101

3、小数部分转二进制 0.4 = 0110 0110 0110 0110 0110...，是无穷尽，取小数跟整数加起来24位即可（**高位1不存入内存**）

4、13.4的二进制表示 13.4 = 1101.0110 0110 0110 0110 0110

5、尾数1.101 0110 0110 0110 0110 0110，左移3，阶符=0

6、阶码=0111 1111（偏移量）+11（左移3） = 1000 0010

7、偏移量作用是把负数转化为整数，这样阶码取值返回就是0~255

|  |  |  |
| --- | --- | --- |
| 0 | 1000 0010 | 101 0110 0110 0110 0110 0110 |

**注意：整数位1不存入内容，固定为1，尾数储存小数部分，所以能多一位来存放小数**

eg2:

double d = 13.4的二级制

过程与float一致，只是精度不一致

|  |  |  |
| --- | --- | --- |
| 0 | 100 0000 0010 | 101 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 |

## 补码运算

**在进行补码运算时，符号位也参与运算，且不考虑符号位的进位问题，直接忽略掉进位。**

### 加法运算

**按补码做加法运算**

1)、计算13+7的值

2)、操作数的补码，13=0000 1101， 7=0000 0111

3)、13+7 = 0000 1101 + 0000 0111 = 0001 0100 = 20

### 减法运算

**转换成加法运算。减去一个数，相当于加上一个负数**

1)、计算13-7的值

2)、13-7 = 13+(-7)

3)、操作数补码，13=0000 1101， -7=1111 1001

4)、13 - 7 = 13+(-7) = 0000 1101 + 1111 1001 = 0000 0110 = 5

### 乘法运算

**补码运算都是通过加法和位移来实现，二进制的乘法运算其实就是被乘数的移位与相加**

补码一位乘法是把乘数Y补码的符号位设成0，当做一个正数，与被乘数补码相乘。乘法运算 的步骤与原码乘法相同。乘积出来后，如果Y是负数，则加上[-X]补得到[X\*Y]补；否则乘积就直接等于[X\*Y]补。这样的补码一位乘法也称校正法。

Eg:12\*11

1100\*1011=1100 + 1 1000 + 00 0000 + 110 0000 =1000 0100=132

Eg:2.5\*0.5

0010.1\*0.1 = 0001.01=1.25

### 除法运算

**二进制的除法运算实际是除数的移位与相减**

**计算机内部是通过计数器记录被除数-除数的次数来求值，**

**Eg：12/6=2**

**1100-0110 = 0110 ---0001**

**0110-0110 = 0000 ---0001**

**所以0001+0001 = 0010 = 2**

# Java基础知识

## 基本类型与对应封装类型

### 基本类型

java基本类型有3类共8种。6个数字类型，1个字符类型，1个布尔类型。**注意String不是基本类型**

**1、byte：8位**有符号以二进制**补码**表示的整数。

最大值：0111 1111 = 2^7-1 = 127

最小值：1111 1111【原码】=1000 0000【反码】=1000 0001【补码】= -(2^7-1)=-127

由于使用补码没有-0=1000 0000，使用-0来表示-128

所以最小值-128

默认值：0

**2、short：16位**有符号以二进制**补码**表示的整数

最大值：0111 1111 1111 1111 = 2^15-1 = 32767

最小值：-32768。原理同byte，负数比整数多一位

默认值：0

**3、int：32位**有符号以二进制**补码**表示的整数

最大值：2^31-1 = 2147483648-1=2147483647

最小值：-2147483648。原理见byte

默认值：0

**4、long：64位**有符号以二进制**补码**表示的整数

最大值：2^63-1

最小值：2^63

默认值：0L

"L"理论上不分大小写，但是若写成"l"容易与数字"1"混淆，不容易分辩。所以最好大写。

**5、float：**float 数据类型是**单精度、32位**、符合IEEE 754标准的浮点数

默认值：0.0f

float a = 5.0f。必须以f结尾，不然会报错

**6、double：**double 数据类型是**双精度、64 位**、符合IEEE 754标准的浮点数

默认值：0.0d

double a = 5.0d或者double a = 5.0。5.0默认是double类型

**用浮点数做计算，可能会损失精度**。如果需要精确计算使用**BigInteger或者BegDecimal**

**7、char：**char类型是一个**单一的 16 位 Unicode 字符**

最大值：\uffff = 2^16-1=65535

最小值：\u0000=0

char 数据类型可以储存任何字符，以Ascii数值来表示字符

char a = 'a'; //字符

int n = 'a'; //ascii

System.out.println(a + ":"+ n);//a:97

**8、boolean：**boolean数据类型表示**一位**的信息

只有两个取值：true 和 false

### 封装类型

基本类型都有对应封装类型

|  |  |  |
| --- | --- | --- |
| 基本类型 | 封装类型 |  |
| int | Integer | Integer.value(int i) |
| long | Long |  |
| byte | Byte |  |
| short | Short |  |
| float | Float |  |
| double | Double |  |
| boolean | Boolean |  |
| char | Character |  |

### 装箱

java从java se5开始提供了自动装箱的机制，从基本类型自动封装成对象类型。

### 拆箱

java从java se5开始提供了自动拆箱的机制，从对象类型自动转成成基本类型

## 引用类型

### 强引用

我们使用的大部分的引用都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

### 软引用

如果一个对象只具有软引用，那就类似于可有可物的生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA虚拟机就会把这个软引用加入到与之关联的引用队列中。

### 弱引用

如果一个对象只具有弱引用，那就类似于可有可物的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

### 虚引用

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

虚引用主要用来跟踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃 圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是 否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

## 运算符

### 算术运算符

### 关系运算符

### 位运算符

### 逻辑运算符

### 赋值运算符

### 条件运算符

### Instanceof运算符

## 数组

## 关键字

## 参数传递

### 值传递

### 引用传递

## IO流

1. File（文件特征与管理）：用于文件或者目录的描述信息，例如生成新目录，修改文件名，删除文件，判断文件所在路径等。

2. InputStream（二进制格式操作）：抽象类，基于字节的输入操作，是所有输入流的父类。定义了所有输入流都具有的共同特征。

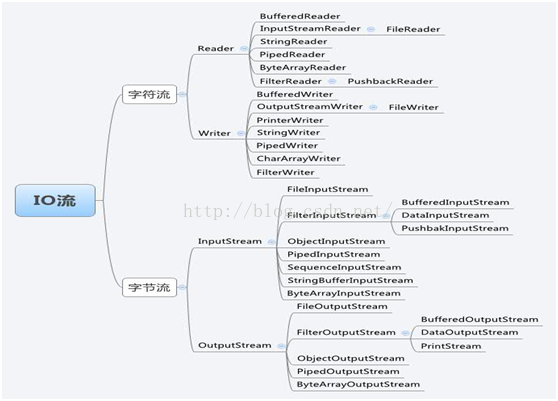
 3. OutputStream（二进制格式操作）：抽象类。基于字节的输出操作。是所有输出流的父类。定义了所有输出流都具有的共同特征。

4.Reader（文件格式操作）：抽象类，基于字符的输入操作。

5. Writer（文件格式操作）：抽象类，基于字符的输出操作。

6. RandomAccessFile（随机文件操作）：一个独立的类，直接继承至Object.它的功能丰富，**可以从文件的任意位置进行存取（输入输出）操作**。

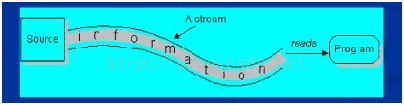




### 输出流、输入流

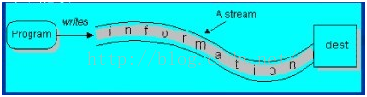
#### 输入流

程序从输入流读取数据源。数据源包括外界(键盘、文件、网络…)，即是将数据源读入到程序的通信通道



#### 输出流

 程序向输出流写入数据。将程序中的数据输出到外界（显示器、打印机、文件、网络…）的通信通道。



#### 特性

 相对于程序来说，输出流是往存储介质或数据通道写入数据，而输入流是从存储介质或数据通道中读取数据，一般来说关于流的特性有下面几点：

1、先进先出，最先写入输出流的数据最先被输入流读取到。

2、顺序存取，可以一个接一个地往流中写入一串字节，读出时也将按写入顺序读取一串字节，不能随机访问中间的数据。（RandomAccessFile**可以从文件的任意位置进行存取（输入输出）操作**）

3、只读或只写，每个流只能是输入流或输出流的一种，不能同时具备两个功能，输入流只能进行读操作，对输出流只能进行写操作。在一个数据传输通道中，如果既要写入数据，又要读取数据，则要分别提供两个流。

### Java IO

在jdk1.3或之前，java只提供阻塞IO编程，java.io包下面。到jdk1.4时才提供非阻塞io编程，在java.nio下面。

阻塞BIO中，网络编程用java.net.Socket和java.net.ServerSocket。**socket通过TCP的三次握手建立连接、四次挥手释放连接**。每个客户端或者每次Socket请求，服务器都会建立一个线程来处理，处理完之后销毁线程，这种模式基本无法处理高并发量的请求。







### Java NIO

与阻塞IO的Socket和ServerSocket对应、非阻塞NIO引入java.nio.channes.SocketChannel和java.nio.channels.ServerSocketChannel。新增两个嵌套字都支持阻塞和非阻塞。

1、缓冲区（Buffer）

传统阻塞IO是面向流的IO，可以将数据直接写入和将数据直接读到stream对象。

非阻塞NIO，所有数据都是用缓冲区来处理。读取数据时，将直接读取到缓冲区中数据，写入数据时，将直接把数据写到缓冲区中。

缓冲区实质是是一个数组，通常是一个字节数组（ByteBuffer），也可以是其他缓冲区。但缓冲区不仅仅是一个数据，还提供了操作缓冲区的数据结构。基本类型中，处理Boolean之外，其他类型都有提供了对应的缓冲区。



2、通道（Channel）

Channel就是一个通道，网络数据通过channel读取和写入。通道（Channel）与流（Stream）不同之处在于通道是双向，流只能在一个方向上移动（一个流必须是InputStream或者OutputStream的子类），通道可以用于读、写或者同时进行。

channel是全双工。

channel可以分为两大类，用于网络读写的SelectableChannel和文件读写FileChannel。

3、多路复用器（Selector）

Selector会不断的轮询注册在其上的Channel，如果每个Channel上面发现了读或写事件，这个Channel就处于就绪状态，会被selector轮询出来，然后通过selectionKey可以或读取就绪的Channel集合，进行后续IO操作。

一个多路复用器Selector可以同时轮询多个Channel，由于JDK使用epoll()代替传统的select实现，所以它并没有最大连接句柄1024/2018的限制。这也就意味着只需要一个线程负责Selector的轮询，就可以接入成千上万的客户端。

### Java AIO

NIO2.0引入新的异步通道概念，并提供异步文件通道和异步嵌套字通道的实现。异步通道提供以下两种方式获取操作结果

1、通过java.util.concurrent.Future来表示异步操作的结果

2、在执行异步操作的时候传入一个java.nio.channels。

CompletionHandler接口的实现类作为操作完成的回调。

NIO2.0的异步嵌套字是真正的异步非阻塞IO，对应Unix网络编程的事件驱动IO（AIO）。它不需要多路复用器（Selector）对注册的通道进行轮询操作即可实现异步读写，从而简化NIO编程模型。

AIO提供了与Socket和ServerSocket对应的java.nio.channes.AsynchronousSocketChannel和java.nio.channels.AsynchronousServerSocketChannel。

### Java 8 Stream

## Java序列化

## 多线程

### 多进程

概述：几乎任何的操作系统都支持运行多个任务，通常一个任务就是一个程序，而一个程序就是一个进程。当一个进程运行时，内部可能包括多个顺序执行流，每个顺序执行流就是一个线程。

 进程：进程是指处于运行过程中的程序，并且具有一定的独立功能。进程是系统进行资源分配和调度的一个单位。当程序进入内存运行时，即为进程。

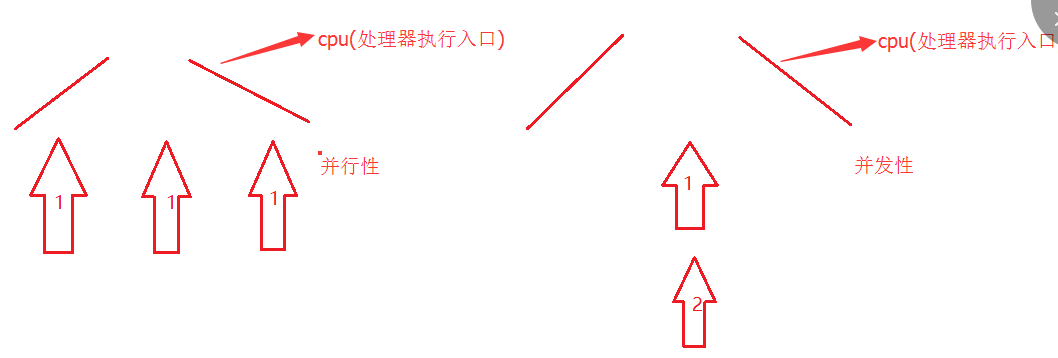
 进程的三个特点：

1：独立性：进程是系统中独立存在的实体，它可以独立拥有资源，每一个进程都有自己独立的地址空间，没有进程本身的运行，用户进程不可以直接访问其他进程的地址空间。

2：动态性：进程和程序的区别在于进程是动态的，进程中有时间的概念，进程具有自己的生命周期和各种不同的状态。

3：并发性：多个进程可以在单个处理器上并发执行，互不影响。

 并发性和并行性是不同的概念：并行是指同一时刻，多个命令在多个处理器上同时执行；并发是指在同一时刻，只有一条命令是在处理器上执行的，但多个进程命令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果



线程：

线程是进程的组成部分，一个进程可以拥有多个线程，而一个线程必须拥有一个父进程。线程可以拥有自己的堆栈，自己的程序计数器和自己的局部变量，但不能拥有系统资源。它与父进程的其他线程共享该进程的所有资源。

线程的特点：

**线程可以完成一定任务，可以和其它线程共享父进程的共享变量和部分环境，相互协作来完成任务。**

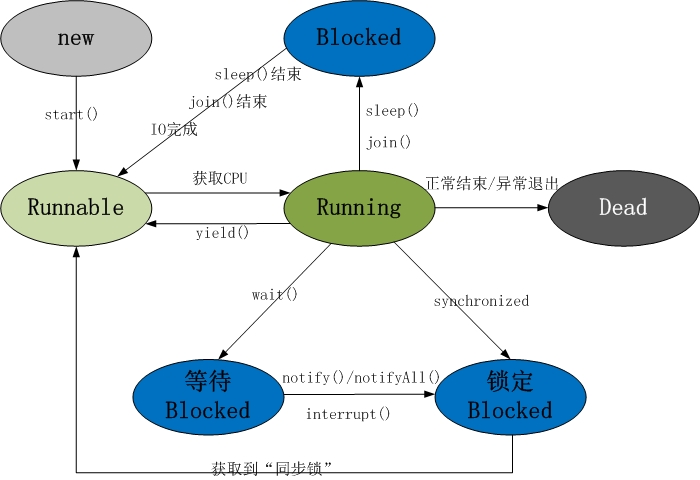
**线程是独立运行的，其不知道进程中是否还有其他线程存在。**

**线程的执行是抢占式的，也就是说，当前执行的线程随时可能被挂起，以便运行另一个线程。**

**一个线程可以创建或撤销另一个线程，一个进程中的多个线程可以并发执行。**

### 多线程

#### 线程的生命周期及五种基本状态



**新建状态（New）：**当线程对象对创建后，即进入了新建状态，如：Thread t = new MyThread();

**就绪状态（Runnable）：**当调用线程对象的start()方法（t.start();），线程即进入就绪状态。处于就绪状态的线程，只是说明此线程已经做好了准备，随时等待CPU调度执行，并不是说执行了t.start()此线程立即就会执行；

**运行状态（Running）：**当CPU开始调度处于就绪状态的线程时，此时线程才得以真正执行，即进入到运行状态。注：就绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

**阻塞状态（Blocked）：**处于运行状态中的线程由于某种原因，暂时放弃对CPU的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才 有机会再次被CPU调用以进入到运行状态。根据阻塞产生的原因不同，阻塞状态又可以分为三种：

1.等待阻塞：运行状态中的线程执行wait()方法，使本线程进入到等待阻塞状态；

2.同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态；

3.其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

**死亡状态（Dead）：**线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

#### 多线程的创建与启动

ava中线程的创建常见有如三种基本形式

**1.继承Thread类，重写该类的run()方法。**

[复制代码](javascript:void(0);)

1 class MyThread extends Thread {

2

3 private int i = 0;

4

5 @Override

6 public void run() {

7 for (i = 0; i < 100; i++) {

8 System.out.println(Thread.currentThread().getName() + " " + i);

9 }

10 }

11 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 public class ThreadTest {

2

3 public static void main(String[] args) {

4 for (int i = 0; i < 100; i++) {

5 System.out.println(Thread.currentThread().getName() + " " + i);

6 if (i == 30) {

7 Thread myThread1 = new MyThread(); // 创建一个新的线程 myThread1 此线程进入新建状态

8 Thread myThread2 = new MyThread(); // 创建一个新的线程 myThread2 此线程进入新建状态

9 myThread1.start(); // 调用start()方法使得线程进入就绪状态

10 myThread2.start(); // 调用start()方法使得线程进入就绪状态

11 }

12 }

13 }

14 }

[复制代码](javascript:void(0);)

如上所示，继承Thread类，通过重写run()方法定义了一个新的线程类MyThread，其中run()方法的方法体代表了线程需要完成的任务，称之为线程执行体。当创建此线程类对象时一个新的线程得以创建，并进入到线程新建状态。通过调用线程对象引用的start()方法，使得该线程进入到就绪状态，此时此线程并不一定会马上得以执行，这取决于CPU调度时机。

**2.实现Runnable接口，并重写该接口的run()方法，该run()方法同样是线程执行体，创建Runnable实现类的实例，并以此实例作为Thread类的target来创建Thread对象，该Thread对象才是真正的线程对象。**

[复制代码](javascript:void(0);)

1 class MyRunnable implements Runnable {

2 private int i = 0;

3

4 @Override

5 public void run() {

6 for (i = 0; i < 100; i++) {

7 System.out.println(Thread.currentThread().getName() + " " + i);

8 }

9 }

10 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 public class ThreadTest {

2

3 public static void main(String[] args) {

4 for (int i = 0; i < 100; i++) {

5 System.out.println(Thread.currentThread().getName() + " " + i);

6 if (i == 30) {

7 Runnable myRunnable = new MyRunnable(); // 创建一个Runnable实现类的对象

8 Thread thread1 = new Thread(myRunnable); // 将myRunnable作为Thread target创建新的线程

9 Thread thread2 = new Thread(myRunnable);

10 thread1.start(); // 调用start()方法使得线程进入就绪状态

11 thread2.start();

12 }

13 }

14 }

15 }

[复制代码](javascript:void(0);)

相信以上两种创建新线程的方式大家都很熟悉了，那么Thread和Runnable之间到底是什么关系呢？我们首先来看一下下面这个例子。

[复制代码](javascript:void(0);)

1 public class ThreadTest {

2

3 public static void main(String[] args) {

4 for (int i = 0; i < 100; i++) {

5 System.out.println(Thread.currentThread().getName() + " " + i);

6 if (i == 30) {

7 Runnable myRunnable = new MyRunnable();

8 Thread thread = new MyThread(myRunnable);

9 thread.start();

10 }

11 }

12 }

13 }

14

15 class MyRunnable implements Runnable {

16 private int i = 0;

17

18 @Override

19 public void run() {

20 System.out.println("in MyRunnable run");

21 for (i = 0; i < 100; i++) {

22 System.out.println(Thread.currentThread().getName() + " " + i);

23 }

24 }

25 }

26

27 class MyThread extends Thread {

28

29 private int i = 0;

30

31 public MyThread(Runnable runnable){

32 super(runnable);

33 }

34

35 @Override

36 public void run() {

37 System.out.println("in MyThread run");

38 for (i = 0; i < 100; i++) {

39 System.out.println(Thread.currentThread().getName() + " " + i);

40 }

41 }

42 }

[复制代码](javascript:void(0);)

同样的，与实现Runnable接口创建线程方式相似，不同的地方在于

1 Thread thread = new MyThread(myRunnable);

那么这种方式可以顺利创建出一个新的线程么？答案是肯定的。至于此时的线程执行体到底是MyRunnable接口中的run()方法还是MyThread类中的run()方法呢？通过输出我们知道线程执行体是MyThread类中的run()方法。其实原因很简单，因为Thread类本身也是实现了Runnable接口，而run()方法最先是在Runnable接口中定义的方法。

1 public interface Runnable {

2

3 public abstract void run();

4

5 }

我们看一下Thread类中对Runnable接口中run()方法的实现：

[复制代码](javascript:void(0);)

　　@Override

public void run() {

if (target != null) {

target.run();

}

}

[复制代码](javascript:void(0);)

也就是说，当执行到Thread类中的run()方法时，会首先判断target是否存在，存在则执行target中的run()方法，也就是实现了Runnable接口并重写了run()方法的类中的run()方法。但是上述给到的列子中，由于多态的存在，根本就没有执行到Thread类中的run()方法，而是直接先执行了运行时类型即MyThread类中的run()方法。

**3.使用Callable和Future接口创建线程。具体是创建Callable接口的实现类，并实现clall()方法。并使用FutureTask类来包装Callable实现类的对象，且以此FutureTask对象作为Thread对象的target来创建线程。**

 看着好像有点复杂，直接来看一个例子就清晰了。

[复制代码](javascript:void(0);)

1 public class ThreadTest {

2

3 public static void main(String[] args) {

4

5 Callable<Integer> myCallable = new MyCallable(); // 创建MyCallable对象

6 FutureTask<Integer> ft = new FutureTask<Integer>(myCallable); //使用FutureTask来包装MyCallable对象

7

8 for (int i = 0; i < 100; i++) {

9 System.out.println(Thread.currentThread().getName() + " " + i);

10 if (i == 30) {

11 Thread thread = new Thread(ft); //FutureTask对象作为Thread对象的target创建新的线程

12 thread.start(); //线程进入到就绪状态

13 }

14 }

15

16 System.out.println("主线程for循环执行完毕..");

17

18 try {

19 int sum = ft.get(); //取得新创建的新线程中的call()方法返回的结果

20 System.out.println("sum = " + sum);

21 } catch (InterruptedException e) {

22 e.printStackTrace();

23 } catch (ExecutionException e) {

24 e.printStackTrace();

25 }

26

27 }

28 }

29

30

31 class MyCallable implements Callable<Integer> {

32 private int i = 0;

33

34 // 与run()方法不同的是，call()方法具有返回值

35 @Override

36 public Integer call() {

37 int sum = 0;

38 for (; i < 100; i++) {

39 System.out.println(Thread.currentThread().getName() + " " + i);

40 sum += i;

41 }

42 return sum;

43 }

44

45 }

[复制代码](javascript:void(0);)

首先，我们发现，在实现Callable接口中，此时不再是run()方法了，而是call()方法，此call()方法作为线程执行体，同时还具有返回值！在创建新的线程时，是通过FutureTask来包装MyCallable对象，同时作为了Thread对象的target。那么看下FutureTask类的定义：

1 public class FutureTask<V> implements RunnableFuture<V> {

2

3 //....

4

5 }

1 public interface RunnableFuture<V> extends Runnable, Future<V> {

2

3 void run();

4

5 }

于是，我们发现FutureTask类实际上是同时实现了Runnable和Future接口，由此才使得其具有Future和Runnable双重特性。通过Runnable特性，可以作为Thread对象的target，而Future特性，使得其可以取得新创建线程中的call()方法的返回值。

执行下此程序，我们发现sum = 4950永远都是最后输出的。而“主线程for循环执行完毕..”则很可能是在子线程循环中间输出。由CPU的线程调度机制，我们知道，“主线程for循环执行完毕..”的输出时机是没有任何问题的，那么为什么sum =4950会永远最后输出呢？

原因在于通过ft.get()方法获取子线程call()方法的返回值时，当子线程此方法还未执行完毕，ft.get()方法会一直阻塞，直到call()方法执行完毕才能取到返回值。

上述主要讲解了三种常见的线程创建方式，对于线程的启动而言，都是调用线程对象的start()方法，需要特别注意的是：**不能对同一线程对象两次调用start()方法。**

### 线程池

#### ThreadPoolExecutor

java.uitl.concurrent.ThreadPoolExecutor类是线程池中最核心的一个类，因此如果要透彻地了解Java中的线程池，必须先了解这个类。下面我们来看一下ThreadPoolExecutor类的具体实现源码。

　　在ThreadPoolExecutor类中提供了四个构造方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class ThreadPoolExecutor extends AbstractExecutorService {      .....      public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,              BlockingQueue<Runnable> workQueue);        public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,              BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory);        public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,              BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler);        public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,          BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler);      ...  } |

 　　从上面的代码可以得知，ThreadPoolExecutor继承了AbstractExecutorService类，并提供了四个构造器，事实上，通过观察每个构造器的源码具体实现，发现前面三个构造器都是调用的第四个构造器进行的初始化工作。

 　　下面解释下一下构造器中各个参数的含义：

* corePoolSize：核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；
* maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；
* keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；
* unit：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

[复制代码](javascript:void(0);)

TimeUnit.DAYS; //天

TimeUnit.HOURS; //小时

TimeUnit.MINUTES; //分钟

TimeUnit.SECONDS; //秒

TimeUnit.MILLISECONDS; //毫秒

TimeUnit.MICROSECONDS; //微妙

TimeUnit.NANOSECONDS; //纳秒

[复制代码](javascript:void(0);)

* workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

ArrayBlockingQueue;

LinkedBlockingQueue;

SynchronousQueue;

　　ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关。

* threadFactory：线程工厂，主要用来创建线程；
* handler：表示当拒绝处理任务时的策略，有以下四种取值：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

 　　具体参数的配置与线程池的关系将在下一节讲述。

　　从上面给出的ThreadPoolExecutor类的代码可以知道，ThreadPoolExecutor继承了AbstractExecutorService，我们来看一下AbstractExecutorService的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public abstract class AbstractExecutorService implements ExecutorService {          protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) { };      protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) { };      public Future<?> submit(Runnable task) {};      public <T> Future<T> submit(Runnable task, T result) { };      public <T> Future<T> submit(Callable<T> task) { };      private <T> T doInvokeAny(Collection<? extends Callable<T>> tasks,                              boolean timed, long nanos)          throws InterruptedException, ExecutionException, TimeoutException {      };      public <T> T invokeAny(Collection<? extends Callable<T>> tasks)          throws InterruptedException, ExecutionException {      };      public <T> T invokeAny(Collection<? extends Callable<T>> tasks,                             long timeout, TimeUnit unit)          throws InterruptedException, ExecutionException, TimeoutException {      };      public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)          throws InterruptedException {      };      public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,                                           long timeout, TimeUnit unit)          throws InterruptedException {      };  } |

 　　AbstractExecutorService是一个抽象类，它实现了ExecutorService接口。

　　我们接着看ExecutorService接口的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public interface ExecutorService extends Executor {        void shutdown();      boolean isShutdown();      boolean isTerminated();      boolean awaitTermination(long timeout, TimeUnit unit)          throws InterruptedException;      <T> Future<T> submit(Callable<T> task);      <T> Future<T> submit(Runnable task, T result);      Future<?> submit(Runnable task);      <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)          throws InterruptedException;      <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,                                    long timeout, TimeUnit unit)          throws InterruptedException;        <T> T invokeAny(Collection<? extends Callable<T>> tasks)          throws InterruptedException, ExecutionException;      <T> T invokeAny(Collection<? extends Callable<T>> tasks,                      long timeout, TimeUnit unit)          throws InterruptedException, ExecutionException, TimeoutException;  } |

 　　而ExecutorService又是继承了Executor接口，我们看一下Executor接口的实现：

|  |  |
| --- | --- |
| 1  2  3 | public interface Executor {      void execute(Runnable command);  } |

 　　到这里，大家应该明白了ThreadPoolExecutor、AbstractExecutorService、ExecutorService和Executor几个之间的关系了。

　　Executor是一个顶层接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable类型，从字面意思可以理解，就是用来执行传进去的任务的；

　　然后ExecutorService接口继承了Executor接口，并声明了一些方法：submit、invokeAll、invokeAny以及shutDown等；

　　抽象类AbstractExecutorService实现了ExecutorService接口，基本实现了ExecutorService中声明的所有方法；

　　然后ThreadPoolExecutor继承了类AbstractExecutorService。

　　在ThreadPoolExecutor类中有几个非常重要的方法：

|  |  |
| --- | --- |
| 1  2  3  4 | execute()  submit()  shutdown()  shutdownNow() |

 　　execute()方法实际上是Executor中声明的方法，在ThreadPoolExecutor进行了具体的实现，这个方法是ThreadPoolExecutor的核心方法，通过这个方法可以向线程池提交一个任务，交由线程池去执行。

　　submit()方法是在ExecutorService中声明的方法，在AbstractExecutorService就已经有了具体的实现，在ThreadPoolExecutor中并没有对其进行重写，这个方法也是用来向线程池提交任务的，但是它和execute()方法不同，它能够返回任务执行的结果，去看submit()方法的实现，会发现它实际上还是调用的execute()方法，只不过它利用了Future来获取任务执行结果（Future相关内容将在下一篇讲述）。

　　shutdown()和shutdownNow()是用来关闭线程池的。

　　还有很多其他的方法：

　　比如：getQueue() 、getPoolSize() 、getActiveCount()、getCompletedTaskCount()等获取与线程池相关属性的方法，有兴趣的朋友可以自行查阅API。

#### 深入剖析线程池的实现原理

在上一节我们从宏观上介绍了ThreadPoolExecutor，下面我们来深入解析一下线程池的具体实现原理，将从下面几个方面讲解：

**1.线程池状态**

**2.任务的执行**

**3.线程池中的线程初始化**

**4.任务缓存队列及排队策略**

**5.任务拒绝策略**

**6.线程池的关闭**

**7.线程池容量的动态调整**

**1.线程池状态**

　　在ThreadPoolExecutor中定义了一个volatile变量，另外定义了几个static final变量表示线程池的各个状态：

|  |  |
| --- | --- |
| 1  2  3  4  5 | volatile int runState;  static final int RUNNING    = 0;  static final int SHUTDOWN   = 1;  static final int STOP       = 2;  static final int TERMINATED = 3; |

 　　runState表示当前线程池的状态，它是一个volatile变量用来保证线程之间的可见性；

　　下面的几个static final变量表示runState可能的几个取值。

　　当创建线程池后，初始时，线程池处于RUNNING状态；

　　如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；

　　如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；

　　当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。

**2.任务的执行**

　　在了解将任务提交给线程池到任务执行完毕整个过程之前，我们先来看一下ThreadPoolExecutor类中其他的一些比较重要成员变量：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | private final BlockingQueue<Runnable> workQueue;              //任务缓存队列，用来存放等待执行的任务  private final ReentrantLock mainLock = new ReentrantLock();   //线程池的主要状态锁，对线程池状态（比如线程池大小                                                                //、runState等）的改变都要使用这个锁  private final HashSet<Worker> workers = new HashSet<Worker>();  //用来存放工作集    private volatile long  keepAliveTime;    //线程存货时间  private volatile boolean allowCoreThreadTimeOut;   //是否允许为核心线程设置存活时间  private volatile int   corePoolSize;     //核心池的大小（即线程池中的线程数目大于这个参数时，提交的任务会被放进任务缓存队列）  private volatile int   maximumPoolSize;   //线程池最大能容忍的线程数    private volatile int   poolSize;       //线程池中当前的线程数    private volatile RejectedExecutionHandler handler; //任务拒绝策略    private volatile ThreadFactory threadFactory;   //线程工厂，用来创建线程    private int largestPoolSize;   //用来记录线程池中曾经出现过的最大线程数    private long completedTaskCount;   //用来记录已经执行完毕的任务个数 |

 　　每个变量的作用都已经标明出来了，这里要重点解释一下corePoolSize、maximumPoolSize、largestPoolSize三个变量。

　　corePoolSize在很多地方被翻译成核心池大小，其实我的理解这个就是线程池的大小。举个简单的例子：

　　假如有一个工厂，工厂里面有10个工人，每个工人同时只能做一件任务。

　　因此只要当10个工人中有工人是空闲的，来了任务就分配给空闲的工人做；

　　当10个工人都有任务在做时，如果还来了任务，就把任务进行排队等待；

　　如果说新任务数目增长的速度远远大于工人做任务的速度，那么此时工厂主管可能会想补救措施，比如重新招4个临时工人进来；

　　然后就将任务也分配给这4个临时工人做；

　　如果说着14个工人做任务的速度还是不够，此时工厂主管可能就要考虑不再接收新的任务或者抛弃前面的一些任务了。

　　当这14个工人当中有人空闲时，而新任务增长的速度又比较缓慢，工厂主管可能就考虑辞掉4个临时工了，只保持原来的10个工人，毕竟请额外的工人是要花钱的。

　　这个例子中的corePoolSize就是10，而maximumPoolSize就是14（10+4）。

　　也就是说corePoolSize就是线程池大小，maximumPoolSize在我看来是线程池的一种补救措施，即任务量突然过大时的一种补救措施。

　　不过为了方便理解，在本文后面还是将corePoolSize翻译成核心池大小。

　　largestPoolSize只是一个用来起记录作用的变量，用来记录线程池中曾经有过的最大线程数目，跟线程池的容量没有任何关系。

　　下面我们进入正题，看一下任务从提交到最终执行完毕经历了哪些过程。

　　在ThreadPoolExecutor类中，最核心的任务提交方法是execute()方法，虽然通过submit也可以提交任务，但是实际上submit方法里面最终调用的还是execute()方法，所以我们只需要研究execute()方法的实现原理即可：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public void execute(Runnable command) {      if (command == null)          throw new NullPointerException();      if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {          if (runState == RUNNING && workQueue.offer(command)) {              if (runState != RUNNING || poolSize == 0)                  ensureQueuedTaskHandled(command);          }          else if (!addIfUnderMaximumPoolSize(command))              reject(command); // is shutdown or saturated      }  } |

 　　上面的代码可能看起来不是那么容易理解，下面我们一句一句解释：

　　首先，判断提交的任务command是否为null，若是null，则抛出空指针异常；

　　接着是这句，这句要好好理解一下：

|  |  |
| --- | --- |
| 1 | if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) |

 　　由于是或条件运算符，所以先计算前半部分的值，如果线程池中当前线程数不小于核心池大小，那么就会直接进入下面的if语句块了。

　　如果线程池中当前线程数小于核心池大小，则接着执行后半部分，也就是执行

|  |  |
| --- | --- |
| 1 | addIfUnderCorePoolSize(command) |

　　如果执行完addIfUnderCorePoolSize这个方法返回false，则继续执行下面的if语句块，否则整个方法就直接执行完毕了。

　　如果执行完addIfUnderCorePoolSize这个方法返回false，然后接着判断：

|  |  |
| --- | --- |
| 1 | if (runState == RUNNING && workQueue.offer(command)) |

 　　如果当前线程池处于RUNNING状态，则将任务放入任务缓存队列；如果当前线程池不处于RUNNING状态或者任务放入缓存队列失败，则执行：

|  |  |
| --- | --- |
| 1 | addIfUnderMaximumPoolSize(command) |

　　如果执行addIfUnderMaximumPoolSize方法失败，则执行reject()方法进行任务拒绝处理。

　　回到前面：

|  |  |
| --- | --- |
| 1 | if (runState == RUNNING && workQueue.offer(command)) |

 　　这句的执行，如果说当前线程池处于RUNNING状态且将任务放入任务缓存队列成功，则继续进行判断：

|  |  |
| --- | --- |
| 1 | if (runState != RUNNING || poolSize == 0) |

 　　这句判断是为了防止在将此任务添加进任务缓存队列的同时其他线程突然调用shutdown或者shutdownNow方法关闭了线程池的一种应急措施。如果是这样就执行：

|  |  |
| --- | --- |
| 1 | ensureQueuedTaskHandled(command) |

 　　进行应急处理，从名字可以看出是保证 添加到任务缓存队列中的任务得到处理。

　　我们接着看2个关键方法的实现：addIfUnderCorePoolSize和addIfUnderMaximumPoolSize：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | private boolean addIfUnderCorePoolSize(Runnable firstTask) {      Thread t = null;      final ReentrantLock mainLock = this.mainLock;      mainLock.lock();      try {          if (poolSize < corePoolSize && runState == RUNNING)              t = addThread(firstTask);        //创建线程去执行firstTask任务          } finally {          mainLock.unlock();      }      if (t == null)          return false;      t.start();      return true;  } |

 　　这个是addIfUnderCorePoolSize方法的具体实现，从名字可以看出它的意图就是当低于核心吃大小时执行的方法。下面看其具体实现，首先获取到锁，因为这地方涉及到线程池状态的变化，先通过if语句判断当前线程池中的线程数目是否小于核心池大小，有朋友也许会有疑问：前面在execute()方法中不是已经判断过了吗，只有线程池当前线程数目小于核心池大小才会执行addIfUnderCorePoolSize方法的，为何这地方还要继续判断？原因很简单，前面的判断过程中并没有加锁，因此可能在execute方法判断的时候poolSize小于corePoolSize，而判断完之后，在其他线程中又向线程池提交了任务，就可能导致poolSize不小于corePoolSize了，所以需要在这个地方继续判断。然后接着判断线程池的状态是否为RUNNING，原因也很简单，因为有可能在其他线程中调用了shutdown或者shutdownNow方法。然后就是执行

|  |  |
| --- | --- |
| 1 | t = addThread(firstTask); |

 　　这个方法也非常关键，传进去的参数为提交的任务，返回值为Thread类型。然后接着在下面判断t是否为空，为空则表明创建线程失败（即poolSize>=corePoolSize或者runState不等于RUNNING），否则调用t.start()方法启动线程。

　　我们来看一下addThread方法的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | private Thread addThread(Runnable firstTask) {      Worker w = new Worker(firstTask);      Thread t = threadFactory.newThread(w);  //创建一个线程，执行任务      if (t != null) {          w.thread = t;            //将创建的线程的引用赋值为w的成员变量          workers.add(w);          int nt = ++poolSize;     //当前线程数加1          if (nt > largestPoolSize)              largestPoolSize = nt;      }      return t;  } |

 　　在addThread方法中，首先用提交的任务创建了一个Worker对象，然后调用线程工厂threadFactory创建了一个新的线程t，然后将线程t的引用赋值给了Worker对象的成员变量thread，接着通过workers.add(w)将Worker对象添加到工作集当中。

　　下面我们看一下Worker类的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64 | private final class Worker implements Runnable {      private final ReentrantLock runLock = new ReentrantLock();      private Runnable firstTask;      volatile long completedTasks;      Thread thread;      Worker(Runnable firstTask) {          this.firstTask = firstTask;      }      boolean isActive() {          return runLock.isLocked();      }      void interruptIfIdle() {          final ReentrantLock runLock = this.runLock;          if (runLock.tryLock()) {              try {          if (thread != Thread.currentThread())          thread.interrupt();              } finally {                  runLock.unlock();              }          }      }      void interruptNow() {          thread.interrupt();      }        private void runTask(Runnable task) {          final ReentrantLock runLock = this.runLock;          runLock.lock();          try {              if (runState < STOP &&                  Thread.interrupted() &&                  runState >= STOP)              boolean ran = false;              beforeExecute(thread, task);   //beforeExecute方法是ThreadPoolExecutor类的一个方法，没有具体实现，用户可以根据              //自己需要重载这个方法和后面的afterExecute方法来进行一些统计信息，比如某个任务的执行时间等              try {                  task.run();                  ran = true;                  afterExecute(task, null);                  ++completedTasks;              } catch (RuntimeException ex) {                  if (!ran)                      afterExecute(task, ex);                  throw ex;              }          } finally {              runLock.unlock();          }      }        public void run() {          try {              Runnable task = firstTask;              firstTask = null;              while (task != null || (task = getTask()) != null) {                  runTask(task);                  task = null;              }          } finally {              workerDone(this);   //当任务队列中没有任务时，进行清理工作          }      }  } |

 　　它实际上实现了Runnable接口，因此上面的Thread t = threadFactory.newThread(w);效果跟下面这句的效果基本一样：

|  |  |
| --- | --- |
| 1 | Thread t = new Thread(w); |

 　　相当于传进去了一个Runnable任务，在线程t中执行这个Runnable。

　　既然Worker实现了Runnable接口，那么自然最核心的方法便是run()方法了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public void run() {      try {          Runnable task = firstTask;          firstTask = null;          while (task != null || (task = getTask()) != null) {              runTask(task);              task = null;          }      } finally {          workerDone(this);      }  } |

 　　从run方法的实现可以看出，它首先执行的是通过构造器传进来的任务firstTask，在调用runTask()执行完firstTask之后，在while循环里面不断通过getTask()去取新的任务来执行，那么去哪里取呢？自然是从任务缓存队列里面去取，getTask是ThreadPoolExecutor类中的方法，并不是Worker类中的方法，下面是getTask方法的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | Runnable getTask() {      for (;;) {          try {              int state = runState;              if (state > SHUTDOWN)                  return null;              Runnable r;              if (state == SHUTDOWN)  // Help drain queue                  r = workQueue.poll();              else if (poolSize > corePoolSize || allowCoreThreadTimeOut) //如果线程数大于核心池大小或者允许为核心池线程设置空闲时间，                  //则通过poll取任务，若等待一定的时间取不到任务，则返回null                  r = workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS);              else                  r = workQueue.take();              if (r != null)                  return r;              if (workerCanExit()) {    //如果没取到任务，即r为null，则判断当前的worker是否可以退出                  if (runState >= SHUTDOWN) // Wake up others                      interruptIdleWorkers();   //中断处于空闲状态的worker                  return null;              }              // Else retry          } catch (InterruptedException ie) {              // On interruption, re-check runState          }      }  } |

 　　在getTask中，先判断当前线程池状态，如果runState大于SHUTDOWN（即为STOP或者TERMINATED），则直接返回null。

　　如果runState为SHUTDOWN或者RUNNING，则从任务缓存队列取任务。

　　如果当前线程池的线程数大于核心池大小corePoolSize或者允许为核心池中的线程设置空闲存活时间，则调用poll(time,timeUnit)来取任务，这个方法会等待一定的时间，如果取不到任务就返回null。

　　然后判断取到的任务r是否为null，为null则通过调用workerCanExit()方法来判断当前worker是否可以退出，我们看一下workerCanExit()的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | private boolean workerCanExit() {      final ReentrantLock mainLock = this.mainLock;      mainLock.lock();      boolean canExit;      //如果runState大于等于STOP，或者任务缓存队列为空了      //或者  允许为核心池线程设置空闲存活时间并且线程池中的线程数目大于1      try {          canExit = runState >= STOP ||              workQueue.isEmpty() ||              (allowCoreThreadTimeOut &&               poolSize > Math.max(1, corePoolSize));      } finally {          mainLock.unlock();      }      return canExit;  } |

 　　也就是说如果线程池处于STOP状态、或者任务队列已为空或者允许为核心池线程设置空闲存活时间并且线程数大于1时，允许worker退出。如果允许worker退出，则调用interruptIdleWorkers()中断处于空闲状态的worker，我们看一下interruptIdleWorkers()的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | void interruptIdleWorkers() {      final ReentrantLock mainLock = this.mainLock;      mainLock.lock();      try {          for (Worker w : workers)  //实际上调用的是worker的interruptIfIdle()方法              w.interruptIfIdle();      } finally {          mainLock.unlock();      }  } |

 　　从实现可以看出，它实际上调用的是worker的interruptIfIdle()方法，在worker的interruptIfIdle()方法中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | void interruptIfIdle() {      final ReentrantLock runLock = this.runLock;      if (runLock.tryLock()) {    //注意这里，是调用tryLock()来获取锁的，因为如果当前worker正在执行任务，锁已经被获取了，是无法获取到锁的                                  //如果成功获取了锁，说明当前worker处于空闲状态          try {      if (thread != Thread.currentThread())      thread.interrupt();          } finally {              runLock.unlock();          }      }  } |

  　　这里有一个非常巧妙的设计方式，假如我们来设计线程池，可能会有一个任务分派线程，当发现有线程空闲时，就从任务缓存队列中取一个任务交给空闲线程执行。但是在这里，并没有采用这样的方式，因为这样会要额外地对任务分派线程进行管理，无形地会增加难度和复杂度，这里直接让执行完任务的线程去任务缓存队列里面取任务来执行。

 　　我们再看addIfUnderMaximumPoolSize方法的实现，这个方法的实现思想和addIfUnderCorePoolSize方法的实现思想非常相似，唯一的区别在于addIfUnderMaximumPoolSize方法是在线程池中的线程数达到了核心池大小并且往任务队列中添加任务失败的情况下执行的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | private boolean addIfUnderMaximumPoolSize(Runnable firstTask) {      Thread t = null;      final ReentrantLock mainLock = this.mainLock;      mainLock.lock();      try {          if (poolSize < maximumPoolSize && runState == RUNNING)              t = addThread(firstTask);      } finally {          mainLock.unlock();      }      if (t == null)          return false;      t.start();      return true;  } |

 　　看到没有，其实它和addIfUnderCorePoolSize方法的实现基本一模一样，只是if语句判断条件中的poolSize < maximumPoolSize不同而已。

　　到这里，大部分朋友应该对任务提交给线程池之后到被执行的整个过程有了一个基本的了解，下面总结一下：

　　1）首先，要清楚corePoolSize和maximumPoolSize的含义；

　　2）其次，要知道Worker是用来起到什么作用的；

　　3）要知道任务提交给线程池之后的处理策略，这里总结一下主要有4点：

* **如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；**
* **如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；**
* **如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；**
* **如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。**

**3.线程池中的线程初始化**

　　默认情况下，创建线程池之后，线程池中是没有线程的，需要提交任务之后才会创建线程。

　　在实际中如果需要线程池创建之后立即创建线程，可以通过以下两个方法办到：

* prestartCoreThread()：初始化一个核心线程；
* prestartAllCoreThreads()：初始化所有核心线程

　　下面是这2个方法的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public boolean prestartCoreThread() {      return addIfUnderCorePoolSize(null); //注意传进去的参数是null  }    public int prestartAllCoreThreads() {      int n = 0;      while (addIfUnderCorePoolSize(null))//注意传进去的参数是null          ++n;      return n;  } |

 　　注意上面传进去的参数是null，根据第2小节的分析可知如果传进去的参数为null，则最后执行线程会阻塞在getTask方法中的

|  |  |
| --- | --- |
| 1 | r = workQueue.take(); |

 　　即等待任务队列中有任务。

**4.任务缓存队列及排队策略**

　　在前面我们多次提到了任务缓存队列，即workQueue，它用来存放等待执行的任务。

　　workQueue的类型为BlockingQueue<Runnable>，通常可以取下面三种类型：

　　1）ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；

　　2）LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE；

　　3）synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

**5.任务拒绝策略**

　　当线程池的任务缓存队列已满并且线程池中的线程数目达到maximumPoolSize，如果还有任务到来就会采取任务拒绝策略，通常有以下四种策略：

|  |  |
| --- | --- |
| 1  2  3  4 | ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。  ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。  ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）  ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务 |

**6.线程池的关闭**

　　ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

* shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务
* shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

**7.线程池容量的动态调整**

　　ThreadPoolExecutor提供了动态调整线程池容量大小的方法：setCorePoolSize()和setMaximumPoolSize()，

* setCorePoolSize：设置核心池大小
* setMaximumPoolSize：设置线程池最大能创建的线程数目大小

　　当上述参数从小变大时，ThreadPoolExecutor进行线程赋值，还可能立即创建新的线程来执行任务。

#### 使用示例

前面我们讨论了关于线程池的实现原理，这一节我们来看一下它的具体使用：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class Test {       public static void main(String[] args) {           ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 200, TimeUnit.MILLISECONDS,                   new ArrayBlockingQueue<Runnable>(5));             for(int i=0;i<15;i++){               MyTask myTask = new MyTask(i);               executor.execute(myTask);               System.out.println("线程池中线程数目："+executor.getPoolSize()+"，队列中等待执行的任务数目："+               executor.getQueue().size()+"，已执行玩别的任务数目："+executor.getCompletedTaskCount());           }           executor.shutdown();       }  }      class MyTask implements Runnable {      private int taskNum;        public MyTask(int num) {          this.taskNum = num;      }        @Override      public void run() {          System.out.println("正在执行task "+taskNum);          try {              Thread.currentThread().sleep(4000);          } catch (InterruptedException e) {              e.printStackTrace();          }          System.out.println("task "+taskNum+"执行完毕");      }  } |

 　　执行结果：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

正在执行task 0

线程池中线程数目：1，队列中等待执行的任务数目：0，已执行玩别的任务数目：0

线程池中线程数目：2，队列中等待执行的任务数目：0，已执行玩别的任务数目：0

正在执行task 1

线程池中线程数目：3，队列中等待执行的任务数目：0，已执行玩别的任务数目：0

正在执行task 2

线程池中线程数目：4，队列中等待执行的任务数目：0，已执行玩别的任务数目：0

正在执行task 3

线程池中线程数目：5，队列中等待执行的任务数目：0，已执行玩别的任务数目：0

正在执行task 4

线程池中线程数目：5，队列中等待执行的任务数目：1，已执行玩别的任务数目：0

线程池中线程数目：5，队列中等待执行的任务数目：2，已执行玩别的任务数目：0

线程池中线程数目：5，队列中等待执行的任务数目：3，已执行玩别的任务数目：0

线程池中线程数目：5，队列中等待执行的任务数目：4，已执行玩别的任务数目：0

线程池中线程数目：5，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

线程池中线程数目：6，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

正在执行task 10

线程池中线程数目：7，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

正在执行task 11

线程池中线程数目：8，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

正在执行task 12

线程池中线程数目：9，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

正在执行task 13

线程池中线程数目：10，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

正在执行task 14

task 3执行完毕

task 0执行完毕

task 2执行完毕

task 1执行完毕

正在执行task 8

正在执行task 7

正在执行task 6

正在执行task 5

task 4执行完毕

task 10执行完毕

task 11执行完毕

task 13执行完毕

task 12执行完毕

正在执行task 9

task 14执行完毕

task 8执行完毕

task 5执行完毕

task 7执行完毕

task 6执行完毕

task 9执行完毕

[复制代码](javascript:void(0);)

　　从执行结果可以看出，当线程池中线程的数目大于5时，便将任务放入任务缓存队列里面，当任务缓存队列满了之后，便创建新的线程。如果上面程序中，将for循环中改成执行20个任务，就会抛出任务拒绝异常了。

　　不过在java doc中，并不提倡我们直接使用ThreadPoolExecutor，而是使用Executors类中提供的几个静态方法来创建线程池：

|  |  |
| --- | --- |
| 1  2  3 | Executors.newCachedThreadPool();        //创建一个缓冲池，缓冲池容量大小为Integer.MAX\_VALUE  Executors.newSingleThreadExecutor();   //创建容量为1的缓冲池  Executors.newFixedThreadPool(int);    //创建固定容量大小的缓冲池 |

 　　下面是这三个静态方法的具体实现;

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public static ExecutorService newFixedThreadPool(int nThreads) {      return new ThreadPoolExecutor(nThreads, nThreads,                                    0L, TimeUnit.MILLISECONDS,                                    new LinkedBlockingQueue<Runnable>());  }  public static ExecutorService newSingleThreadExecutor() {      return new FinalizableDelegatedExecutorService          (new ThreadPoolExecutor(1, 1,                                  0L, TimeUnit.MILLISECONDS,                                  new LinkedBlockingQueue<Runnable>()));  }  public static ExecutorService newCachedThreadPool() {      return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,                                    60L, TimeUnit.SECONDS,                                    new SynchronousQueue<Runnable>());  } |

　　从它们的具体实现来看，它们实际上也是调用了ThreadPoolExecutor，只不过参数都已配置好了。

　　newFixedThreadPool创建的线程池corePoolSize和maximumPoolSize值是相等的，它使用的LinkedBlockingQueue；

　　newSingleThreadExecutor将corePoolSize和maximumPoolSize都设置为1，也使用的LinkedBlockingQueue；

　　newCachedThreadPool将corePoolSize设置为0，将maximumPoolSize设置为Integer.MAX\_VALUE，使用的SynchronousQueue，也就是说来了任务就创建线程运行，当线程空闲超过60秒，就销毁线程。

　　实际中，如果Executors提供的三个静态方法能满足要求，就尽量使用它提供的三个方法，因为自己去手动配置ThreadPoolExecutor的参数有点麻烦，要根据实际任务的类型和数量来进行配置。

　　另外，如果ThreadPoolExecutor达不到要求，可以自己继承ThreadPoolExecutor类进行重写。

#### 如何合理配置线程池大小

本节来讨论一个比较重要的话题：如何合理配置线程池大小，仅供参考。

　　一般需要根据任务的类型来配置线程池大小：

　　如果是CPU密集型任务，就需要尽量压榨CPU，参考值可以设为 NCPU+1

　　如果是IO密集型任务，参考值可以设置为2\*NCPU

　　当然，这只是一个参考值，具体的设置还需要根据实际情况进行调整，比如可以先将线程池大小设置为参考值，再观察任务运行情况和系统负载、资源利用率来进行适当调整。

## 对象池

对象的创建和销毁在一定程度上会消耗系统的资源，虽然jvm的性能在近几年已经得到了很大的提高，对于多数对象来说，没有必要利用对象池技术来进行对象的创建和管理。但是对于有些对象来说，其创建的代价还是比较昂贵的，比如线程、tcp连接、rpc连接、数据库连接等对象，因此对象池技术还是有其存在的意义。

Apache-commons-pool-1.6提供的对象池主要有两种：一种是带Key的对象池，这种带Key的对象池是把相同的池对象放在同一个池中，也就是说有多少个key就有多少个池；另一种是不带Key的对象池，这种对象池是把生产完全一致的对象放在同一个池中，但是有时候，单用对池内所有对象一视同仁的对象池，并不能解决的问题。例如：对于一组某些参数设置不同的同类对象——比如一堆指向不同地址的 java.net.URL对象或者一批代表不同语句的java.sql.PreparedStatement对象，用这样的方法池化，就有可能取出不合用的对象。

1、对象池：

1）对象池接口介绍：

如果让我们去设计一个对象池接口，会给用户提供哪些核心的方法呢？

borrowObject()，returnObject()是两个核心方法，一个是’借’，一个是’还’。那么我们有可能需要对一个已经借到的对象置为失效（比如当我们的远程连接关闭或产生异常，这个连接不可用需要失效掉），invalidateObject()也是必不可少的。对象池刚刚创建的时候，我们可能需要预热一部分对象，而不是采用懒加载模式以避免系统启动时候的抖动，因此addObject()提供给用户，以进行对象池的预热。有创建就有销毁，clear()和close()就是用来清空对象池（觉得叫purge()可能更好一点)。除此之外，我们可能还需要一些简单的统计，比如getNumIdle()获得空闲对象个数和getNumActive()获得活动对象（被借出对象）的个数。如下表：

方法名 作用

borrowObject() 从池中借对象

returnObject() 还回池中

invalidateObject() 失效一个对象

addObject() 池中增加一个对象

clear() 清空对象池

close() 关闭对象池

getNumIdle() 获得空闲对象数量

getNumActive() 获得被借出对象数量

2）在commons-pool中有两类对象池接口（带key和不带key），一个是ObjectPool，另一个是KeyedObjectPool；此外，为了方便他们分别还对应了ObjectPoolFactory、KeyedObjectPoolFactory两个接口（这两个接口在功能上和他们都一样，只是使用形式上不一样）

3）对象池空间划分：

一个对象存储到对象池中，其位置不是一成不变的。空间的划分可以分为两种，一种是物理空间划分，一种是逻辑空间划分。不同的实现可能采用不同的技术手段，Commons Pool实际上采用了逻辑划分。如下图所示：

从整体上来讲，可以将空间分为池外空间和池内空间，池外空间是指被’出借’的对象所在的空间（逻辑空间）。池内空间进一步可以划分为idle空间，abandon空间和invalid空间。idle空间就是空闲对象所在的空间，空闲对象之间是有一定的组织结构的（详见后文）。abandon空间又被称作放逐空间，用于放逐被出借超时的对象。invalid空间其实就是对象的垃圾场，这些对象将不会在被使用，而是等待被gc处理掉。

4）对象池的放逐与驱逐策略：

下面我们会多次提到驱逐(eviction)和放逐(abandon)，这两个概念是对象池设计的核心。

       先来看驱逐，我们知道对象池的一个重要的特性就是伸缩性，所谓伸缩性是指对象池能够根据当前池中空闲对象的数量（maxIdle和minIdle配置）自动进行调整，进而避免内存的浪费。自动伸缩，这是驱逐所需要达到的目标，他是如何实现的呢？实际上在对象池内部，我们可以维护一个驱逐定时器(EvictionTimer)，由timeBetweenEvictionRunsMillis参数对定时器的间隔加以控制，每次达到驱逐时间后，我们就选定一批对象（由numTestsPerEvictionRun参数进行控制）进行驱逐测试，这个测试可以采用策略模式，比如Commons Pool的DefaultEvictionPolicy，代码如下：

@Override

public boolean evict(EvictionConfig config, PooledObject<T> underTest,

int idleCount) {

if ((config.getIdleSoftEvictTime() < underTest.getIdleTimeMillis() &&

config.getMinIdle() < idleCount) ||

config.getIdleEvictTime() < underTest.getIdleTimeMillis()) {

return true;

}

return false;

}

对于符合驱逐条件的对象，将会被对象池无情的驱逐出空闲空间，并丢弃到invalid空间。之后对象池还需要保证内部空闲对象数量需要至少达到minIdle的控制要求。

       我们在看来放逐，对象出借时间太长（由removeAbandonedTimeout控制），我们就把他们称作流浪对象，这些对象很有可能是那些用完不还的坏蛋们的杰作，也有可能是对象使用者出现了什么突发状况，比如网络连接超时时间设置长于放逐时间。总之，被放逐的对象是不允许再次回归到对象池中的，他们会被搁置到abandon空间，进而进入invalid空间再被gc掉以完成他们的使命。放逐由removeAbandoned()方法实现，分为标记过程和放逐过程，代码实现并不难，有兴趣的可以直接翻翻源代码。

驱逐是由内而外将对象驱逐出境，放逐则是由外而内，将对象流放。他们一内一外，正是整个对象池形成闭环的核心要素。

5）对象池有效性探测：

用过数据库连接池的同学可能对类似testOnBorrow的配置比较熟悉。除了testOnBorrow，对象池还提供了testOnCreate, testOnReturn, testWhileIdle，其中testWhileIdle是当对象处于空闲状态的时候所进行的测试，当测试通过则继续留在对象池中，如果失效，则弃置到invalid空间。所谓testOnBorrow其实就是当对象出借前进行测试，测试什么？当然是有效性测试，在测试之前我们需要调用factory.activateObject()以激活对象，在调用factory.validateObject(p)对准备出借的对象做有有效性检查，如果这个对象无效则可能有抛出异常的行为，或者返回空对象，这全看具体实现了。testOnCreate表示当对象创建之后，再进行有效性测试，这并不适用于频繁创建和销毁对象的对象池，他与testOnBorrow的行为类似。testOnReturn是在对象还回到池子之前锁进行的测试，与出借的测试不同，testOnReturn无论是测试成功还是失败，我们都需要保证池子中的对象数量是符合配置要求的()ensureIdle()方法就是做这个事情)，并且如果测试失败了，我们可以直接swallow这个异常，因为用户根本不需要关心池子的状态。

6）对象池的常见配置一览：

配置参数 意义 默认值

maxTotal 对象总数 8

maxIdle 最大空闲对象数 8

minIdle 最小空闲对象书 0

lifo 对象池借还是否采用lifo true

fairness 对于借对象的线程阻塞恢复公平性 false

maxWaitMillis 借对象阻塞最大等待时间 -1

minEvictableIdleTimeMillis 最小驱逐空闲时间 30分钟

numTestsPerEvictionRun 每次驱逐数量 3

testOnCreate 创建后有效性测试 false

testOnBorrow 出借前有效性测试 false

testOnReturn 还回前有效性测试 false

testWhileIdle 空闲有效性测试 false

timeBetweenEvictionRunsMillis 驱逐定时器周期 false

blockWhenExhausted 对象池耗尽是否block true

2、池化对象：

1）池化对象接口：（被池化的对象需要实现该接口）

池化对象就是对象池中所管理的基本单元。我们可以思考一下，如果直接将我们的原始对象放到对象池中是否可以？答案当然是可以，但是不好，因为如果那样做，我们的对象池就退化成了容器Collection了，之所以需要将原始对象wrapper成池对象，是因为我们需要提供额外的管理功能，比如生命周期管理。commons pool采用了PooledObject<T>接口和KeyedPooledObject<T>接口用于表达池对象，它主要抽象了池对象的状态管理和一些诸如状态变迁时所产生的统计指标，这些指标可以配合对象池做更精准的管理操作。

2）池化对象状态：

说到对池对象的管理，最重要的当属对状态的管理。对于状态管理，我们熟知的模型就是状态机模型了。池对象当然也有一套自己的状态机，我们先来看看commons pool所定义的池对象都有哪些状态：

状态 解释

IDLE 空闲状态

ALLOCATED 已出借状态

EVICTION 正在进行驱逐测试

EVICTION\_RETURN\_TO\_HEAD 驱逐测试通过对象放回到头部

VALIDATION 空闲校验中

VALIDATION\_PREALLOCATED 出借前校验中

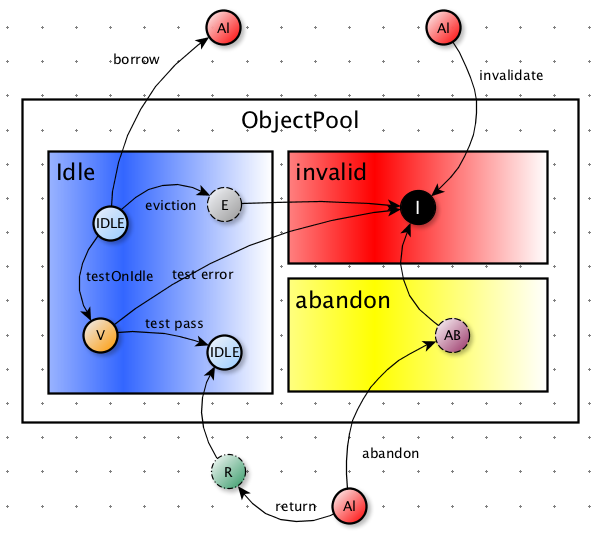
VALIDATION\_RETURN\_TO\_HEAD 校验通过后放回头部

INVALID 无效对象

ABANDONED 放逐中

RETURNING 换回对象池中

这里只需知道：放逐（ABANDONED）指的是不在对象池中的对象超时流放；驱逐（EVICTION）指的是空闲对象超时销毁；VALIDATION是有效性校验，主要校验空闲对象的有效性。注意与驱逐和放逐之间的区别。我们通过一张图来看看状态之间的变迁。



我们看到上图的’圆圈’表示的就是池对象，其中中间的英文简写是其对应的状态。虚线外框则表示瞬时状态。比如RETURNING和ABANDONED。这里我们省略了VALIDATION\_RETURN\_TO\_HEAD，VALIDATION\_PREALLOCATED，EVICTION\_RETURN\_TO\_HEAD，因为这对于我们理解池对象状态变迁并没有太多帮助。针对上图，我们重点关注四个方面：

IDLE->ALLOCATED 即上图的borrow操作，除了需要将状态置为已分配，我们还需要考虑如果对象池耗尽了怎么办？是继续阻塞还是直接异常退出？如果阻塞是阻塞多久？

ALLOCATED->IDLE 即上图的return操作，我们需要考虑的是，如果池对象还回到对象池，此时对象池空闲数已经达到上界或该对象已经无效，我们是否需要进行特殊处理？

IDLE->EVICTION 与 ALLOCATED->ABANDONED 请参考后文

IDLE->VALIDATION 是testWhileIdle的有效性测试所需要经历的状态变迁，他是指每隔一段时间对池中所有的idle对象进行有效性检查，以排除那些已经失效的对象。失效的对象将会弃置到invalid空间。

3）池化对象生命周期控制：

只搞清楚了池化对象的状态和状态转移是不够的，我们还应该能够对池对象生命周期施加影响。Commons Pool通过PooledObjectFactory<T>接口和KeyedPooledObjectFactory<T>对对象生命周期进行控制。该接口有如下方法：

方法 解释

makeObject 创建对象

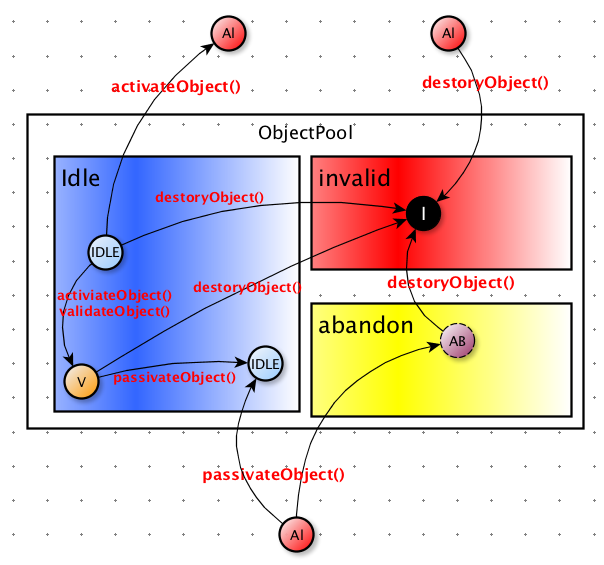
destroyObject 销毁对象

validateObject 校验对象

activateObject 重新初始化对象

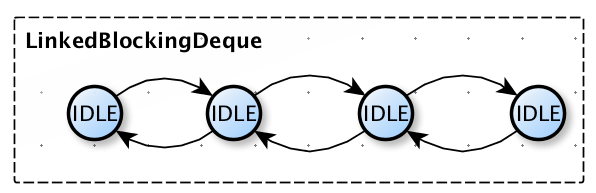
passivateObject 反初始化对象

我们需要注意，池对象必须经过创建（makeObject()）和初始化过程（activateObject()）后才能够被我们使用。我们看一看这些方法能够影响哪些状态变迁。



4）池对象组织结构：

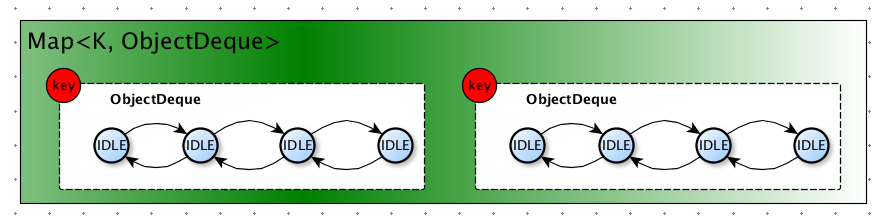
池中的对象，并不是杂乱无章的，他们得有一定的组织结构。不同的组织结构可能会从整体影响对象池的使用。Apache Commons提供了两种组织结构，其一是有界阻塞双端队列(LinkedBlockingDeque)，其二是key桶。



有界阻塞队列能够提供阻塞特性，当池中对象exhausted后，新申请对象的线程将会阻塞，这是典型的生产者/消费者模型，通过这种双端的阻塞队列，我们能够实现池对象的lifo或fifo。如下代码：

|  |
| --- |
| if (getLifo()) {  idleObjects.addFirst(p);  } else {  idleObjects.addLast(p);  } |

因为是带有阻塞性质的队列，我们能够通过fairness参数控制线程获得锁的公平性，这里我们可以参考AQS实现，不说了。下面我们再来看一看key桶的数据结构：



从上图我们可以看出，每一个key对应一个的双端阻塞队列ObjectDeque，ObjectDeque实际上就是包装了LinkedBlockingDeque，采用这种结构我们能够对池对象进行一定的划分，从而更加灵活的使用对象池。Commons Pool采用了KeyedObjectPool<K,V>用以表示采用这种数据结构的对象池。当我们borrow和return的时候，都需要指定对应的key空间。

## Java.util.Collection类

### List

#### ArrayList

#### LinkedList

#### Vector

#### SynchronizedList

### Set

#### HashSet

#### TreeSet

#### LinkedHashSet

#### PersistentSet

#### PredicatedSet

#### SortedSet

### Queue

#### ConcurrentLinkedQueue

#### BlockingQueue

#### Deque

## Java.util.Map类型

### HashMap

### HashTable

### SoftHashMap

### WeakHashMap

### SortedMap

## Java.util.Date

## Java.util.concurrent包

### ConcurrentHashMap

### ConcurrentLinkedQueue

### ConcurrentLinkedDeque

## Jdk版本

## Java8

### Lambda

|  |
| --- |
| **package** org.xzh.springboot.java8.lambda;  @FunctionalInterface  **public** **interface** LoggerInfo {  **public** **void** info(String name);  } |

|  |
| --- |
| **package** org.xzh.springboot.java8.lambda;  **public** **class** AppleInfo {  **public** **void** printSize(**int** size, LoggerInfo logger){  logger.info(size+"cm");  }    **public** **void** printWeight(**int** weight, LoggerInfo logger){  logger.info(weight+"kg");  }  } |

|  |
| --- |
| **package** org.xzh.springboot.java8.lambda;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** LambdaMain {    **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(LambdaMain.**class**);  **public** **static** **void** main(String[] args) {  AppleInfo appleInfo = **new** AppleInfo();  appleInfo.printSize(12, msg -> ***logger***.info(msg));  appleInfo.printWeight(1, msg -> System.***out***.println(msg));  }  } |

### Stream

|  |
| --- |
| **package** org.xzh.springboot.java8.steam;  **public** **class** Dish {  **private** **final** String name;  **private** **final** **boolean** vegetarian;  **private** **final** **int** calories;  **private** **final** Type type;  **public** Dish(String name, **boolean** vegetarian, **int** calories, Type type) {  **this**.name = name;  **this**.vegetarian = vegetarian;  **this**.calories = calories;  **this**.type = type;  }  **public** String getName() {  **return** name;  }  **public** **boolean** isVegetarian() {  **return** vegetarian;  }  **public** **int** getCalories() {  **return** calories;  }  **public** Type getType() {  **return** type;  }  @Override  **public** String toString() {  **return** name;  }  **public** **enum** Type {  ***MEAT***, ***FISH***, ***OTHER***  }  } |

|  |
| --- |
| **package** org.xzh.springboot.java8.steam;  **import** java.util.Arrays;  **import** java.util.List;  **import** java.util.Optional;  **import** java.util.stream.Collectors;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** StreamMain {    **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(StreamMain.**class**);  **public** **static** **void** main(String[] args) {  List<Dish> menu = Arrays.*asList*(  **new** Dish("pork", **false**, 800, Dish.Type.***MEAT***),  **new** Dish("beef", **false**, 700, Dish.Type.***MEAT***),  **new** Dish("chicken", **false**, 400, Dish.Type.***MEAT***),  **new** Dish("french fries", **true**, 530, Dish.Type.***OTHER***),  **new** Dish("rice", **true**, 350, Dish.Type.***OTHER***),  **new** Dish("season fruit", **true**, 120, Dish.Type.***OTHER***),  **new** Dish("pizza", **true**, 550, Dish.Type.***OTHER***),  **new** Dish("prawns", **false**, 300, Dish.Type.***FISH***),  **new** Dish("salmon", **false**, 450, Dish.Type.***FISH***) );    //筛选:用谓词筛选,Streams接口支持filter方法（你现在应该很熟悉了）。该操作会接受一个谓词（一个返回  //boolean的函数）作为参数，并返回一个包括所有符合谓词的元素的流。  List<Dish> filterDish = menu.stream().filter(Dish::isVegetarian).collect(Collectors.*toList*());  ***logger***.info("filter dish is {}", filterDish);    //筛选各异的元素.  //流还支持一个叫作distinct的方法，它会返回一个元素各异（根据流所生成元素的  //hashCode和equals方法实现）的流。  List<Integer> numbers = Arrays.*asList*(1, 2, 1, 3, 3, 2, 4);  numbers.stream().filter(i->i%2==0).distinct().forEach(msg -> ***logger***.info(msg+""));    //截短流  //流支持limit(n)方法，该方法会返回一个不超过给定长度的流。所需的长度作为参数传递  //给limit。如果流是有序的，则最多会返回前n个元素。  List<Dish> limitDish = menu.stream().filter(p -> p.getCalories()>300).limit(3).collect(Collectors.*toList*());  ***logger***.info("filter dish is {}", limitDish);    //跳过元素  //流还支持skip(n)方法，返回一个扔掉了前n个元素的流。如果流中元素不足n个，则返回一  //个空流。请注意，limit(n)和skip(n)是互补的  List<Dish> skipDish = menu.stream().filter(p -> p.getCalories()>300).skip(2).collect(Collectors.*toList*());  ***logger***.info("skip dish is {}", skipDish);      //-----------------------------------------------------    //映射  List<String> names = menu.stream().map(p -> p.getName()).collect(Collectors.*toList*());  ***logger***.info("map dish is {}",names);      //---------------------------------------------------------    //查找与匹配  //检查谓词是否至少匹配一个元素  **if**(menu.stream().anyMatch(p->p.isVegetarian())){  ***logger***.info("anyMatch is true");  }    //检查谓词是否匹配所有元素  //allMatch方法的工作原理和anyMatch类似，但它会看看流中的元素是否都能匹配给定的谓词  //和allMatch相对的是noneMatch。它可以确保流中没有任何元素与给定的谓词匹配  ***logger***.info("allMatch is {}", menu.stream().allMatch(p->p.isVegetarian()));  ***logger***.info("allMatch is {}", menu.stream().noneMatch(p->p.isVegetarian()));    //findAny方法将返回当前流中的任意元素。  Optional<Dish> findAnyDish = menu.stream().filter(p->p.isVegetarian()).findAny();  ***logger***.info(findAnyDish.get().getName());    //查找第一个元素  Optional<Dish> findFirstDish = menu.stream().filter(p->p.isVegetarian()).findFirst();  **if**(findFirstDish.isPresent()){  ***logger***.info(findFirstDish.get().getName());  }      //------------------------------------------------    //归约  //元素求和  **int** sum = menu.stream().mapToInt(p->p.getCalories()).reduce(0, (a, b)->a+b);  ***logger***.info("sum {}", sum);  Optional<Integer> sum2 = menu.stream().map(p -> p.getCalories()).reduce((a, b) -> a+b);  **if**(sum2.isPresent()){  ***logger***.info("sum2 {}", sum2.get());  }    //最大值和最小值  Optional<Integer> min = menu.stream().map(p->p.getCalories()).reduce((a, b)->Integer.*min*(a, b));  **if**(min.isPresent()){  ***logger***.info("min {}", min.get());  }  Optional<Integer> max = menu.stream().map(p->p.getCalories()).reduce((a, b)-> a>b?a:b);  **if**(max.isPresent()){  ***logger***.info("max {}", max.get());  }  }  } |

### 函数式接口

只有一个没有默认实现方法的接口，叫做函数式接口

|  |
| --- |
| **package** org.xzh.springboot.java8;  @FunctionalInterface  **public** **interface** FunctionInterfaceTest {  **public** **void** logger(String name);    **default** **int** sum(**int** a, **int** b){  **return** a+b;  }  } |

## 反射机制

|  |
| --- |
| **package** org.xzh.springboot.reflect;  **import** java.lang.reflect.Constructor;  **import** java.lang.reflect.Field;  **import** java.lang.reflect.InvocationTargetException;  **import** java.lang.reflect.Method;  **import** org.junit.Test;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** ReflectTest {  **private** **static** **final** Logger ***logger*** = LoggerFactory  .*getLogger*(ReflectTest.**class**);  **private** **static** Class<User> *userClass* = User.**class**;  /\*\*  \* 在类加载的时候，jvm会创建一个class对象  \*  \* class对象是可以说是反射中最常用的，获取class对象的方式的主要有三种  \*  \* 根据类名：类名.class 根据对象：对象.getClass() 根据全限定类名：Class.forName(全限定类名)  \*  \* **@throws** Exception  \*/  @Test  **public** **void** classTest() **throws** Exception {  // 获取class对象的三种方法  Class<?> clazz1 = User.**class**;  ***logger***.info("根据类名：{}", clazz1);  Class<?> clazz2 = **new** User().getClass();  ***logger***.info("根据对象：{}", clazz2);  Class<?> clazz3 = Class.*forName*("org.xzh.springboot.reflect.User");  ***logger***.info("根据全限定类名：{}", clazz3);  // class常用方法  ***logger***.info("获取全限定类名:{}", *userClass*.getName());  ***logger***.info("获取类名：{}", *userClass*.getSimpleName());  ***logger***.info("实例化：{}", *userClass*.newInstance());  }  /\*\*  \* 构造函数是java创建对象的必经之路，所以通过反射拿到一个类的构造函数后， 再去创建这个类的对象自然是易如反掌，  \* 常用的方法如下：  \*  \* **@throws** InvocationTargetException  \* **@throws** IllegalArgumentException  \* **@throws** IllegalAccessException  \* **@throws** InstantiationException  \*/  @Test  **public** **void** constructorTest() **throws** InstantiationException,  IllegalAccessException, IllegalArgumentException,  InvocationTargetException {  // 获取全部的构造函数  Constructor<?>[] constructors = *userClass*.getConstructors();  // 取消安全性检查,设置后才可以使用private修饰的构造函数，也可以单独对某个构造函数进行设置  // Constructor.setAccessible(constructors, true);  **for** (**int** i = 0; i < constructors.length; i++) {  Constructor<?> constructor = constructors[i];  // 获取构造方法参数类型  Class<?>[] types = constructor.getParameterTypes();  ***logger***.info("第{}个构造方法，参数{}", i, types);  **if** (types.length == 0) {  ***logger***.info("第{}个构造方法，实例化{}", i, constructor.newInstance());  } **else** {  ***logger***.info("第{}个构造方法，实例化{}", i,  constructor.newInstance("zhang", "18"));  }  }  }  /\*\*  \* 犹记得学习spring ioc之时，对未提供set方法的private属性依然可以注入感到神奇万分，  \* 现在看来，这神奇的根源自然是来自于java的反射，常用的方法如下：  \*  \* **@throws** IllegalAccessException  \* **@throws** InstantiationException  \* **@throws** SecurityException  \* **@throws** NoSuchFieldException  \*/  @Test  **public** **void** fieldTest() **throws** InstantiationException,  IllegalAccessException, NoSuchFieldException, SecurityException {  User user = *userClass*.newInstance();  // 获取当前类所有属性  Field[] fields = *userClass*.getDeclaredFields();  // 获取公有属性(包括父类)  // Field fields[] = cl.getFields();  // 取消安全性检查,设置后才可以获取或者修改private修饰的属性，也可以单独对某个属性进行设置  Field.*setAccessible*(fields, **true**);  **for** (Field field : fields) {  ***logger***.info("属性名称：{}，属性值：{}，属性类型：{}", field.getName(),  field.get(user), field.getType());  }  fields[0].set(user, "zhangsan");  fields[1].set(user, "13");  ***logger***.info("实例化{}", user);  Field nameField = *userClass*.getDeclaredField("name");  // 取消安全性检查,设置后才可以获取或者修改private修饰的属性，也可以批量对所有属性进行设置  nameField.setAccessible(**true**);  nameField.set(user, "lisi");  ***logger***.info("实例化{}", user);  }  /\*\*  \* 大家对javabean肯定不会陌生，在用框架操作javabean时，  \* 大多都是通过反射调用get,set方法Javabean进行操作，常用的方法如下：  \* **@throws** IllegalAccessException  \* **@throws** InstantiationException  \* **@throws** SecurityException  \* **@throws** NoSuchMethodException  \* **@throws** InvocationTargetException  \* **@throws** IllegalArgumentException  \*/  @Test  **public** **void** methodTest() **throws** InstantiationException, IllegalAccessException, NoSuchMethodException, SecurityException, IllegalArgumentException, InvocationTargetException {  User user = *userClass*.newInstance();  // 获取当前类的所有方法  Method[] methods = *userClass*.getDeclaredMethods();  // 获取公有方法(包括父类)  // Method[] methods = userClass.getMethods();  // 取消安全性检查,设置后才可以调用private修饰的方法，也可以单独对某个方法进行设置  Method.*setAccessible*(methods, **true**);    **for**(Method method : methods){  ***logger***.info("方法名称{}，参数{}， 返回类型{}", method.getName(),method.getParameters(), method.getReturnType().getName());  }    // 获取无参方法  Method getNameMethod = *userClass*.getDeclaredMethod("getName");  // 取消安全性检查,设置后才可以调用private修饰的方法，也可以批量对所有方法进行设置  getNameMethod.setAccessible(**true**);  ***logger***.info("调用getName方法：{}", getNameMethod.invoke(user));    // 获取有参方法  Method setNameMethod = *userClass*.getDeclaredMethod("setName", String.**class**);  setNameMethod.setAccessible(**true**);  setNameMethod.invoke(user, "lisis");  ***logger***.info("实例化{}", user);  }  } |

## 注解

### 认识注解

注解(Annotation)很重要，未来的开发模式都是基于注解的，JPA是基于注解的，Spring2.5以上都是基于注解的，Hibernate3.x以后也是基于注解的，现在的Struts2有一部分也是基于注解的了，注解是一种趋势，现在已经有不少的人开始用注解了，注解是JDK1.5之后才有的新特性

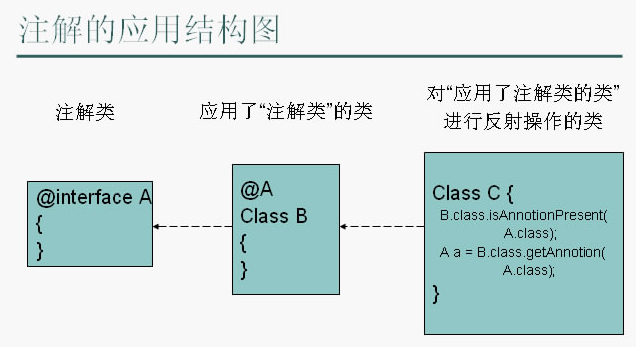
JDK1.5之后内部提供的三个注解

       @Deprecated 意思是“**废弃的，过时的**”

       @Override 意思是“**重写、覆盖**”

       @SuppressWarnings 意思是“**压缩警告**”

总结：**注解(Annotation)相当于一种标记，在程序中加入注解就等于为程序打上某种标记**，没有加，则等于没有任何标记，以后，javac编译器、开发工具和其他程序可以通过反射来了解你的类及各种元素上有无何种标记，看你的程序有什么标记，就去干相应的事，**标记可以加在包、类，属性、方法，方法的参数以及局部变量上。**



### @Retention元注解

根据反射的测试的问题，引出@Retention元注解的讲解：其三种取值：**RetentionPolicy.SOURCE**、**RetentionPolicy.CLASS**、**RetentionPolicy.RUNTIME**分别对应：Java源文件(.java文件)---->.class文件---->内存中的字节码

当在Java源程序上加了一个注解，这个Java源程序要由javac去编译，javac把java源文件编译成.class文件，在编译成class时可能会把Java源程序上的一些注解给去掉，java编译器(javac)在处理java源程序时，可能会认为这个注解没有用了，于是就把这个注解去掉了，那么此时在编译好的class中就找不到注解了， 这是编译器编译java源程序时对注解进行处理的第一种可能情况，假设java编译器在把java源程序编译成class时，没有把java源程序中的注解去掉，那么此时在编译好的class中就可以找到注解，当程序使用编译好的class文件时，需要用类加载器把class文件加载到内存中，class文件中的东西不是字节码，class文件里面的东西由类加载器加载到内存中去，类加载器在加载class文件时，会对class文件里面的东西进行处理，如安全检查，处理完以后得到的最终在内存中的二进制的东西才是字节码，类加载器在把class文件加载到内存中时也有转换，转换时是否把class文件中的注解保留下来，这也有说法，所以说**一个注解的生命周期有三个阶段：java源文件是一个阶段，class文件是一个阶段，内存中的字节码是一个阶段**,javac把java源文件编译成.class文件时，有可能去掉里面的注解，类加载器把.class文件加载到内存时也有可能去掉里面的注解，因此**在自定义注解时就可以使用Retention注解指明自定义注解的生命周期，自定义注解的生命周期是在RetentionPolicy.SOURCE阶段(java源文件阶段)，还是在RetentionPolicy.CLASS阶段(class文件阶段)，或者是在RetentionPolicy.RUNTIME阶段(内存中的字节码运行时阶段)**，根据**JDK提供的API可以知道默认是在RetentionPolicy.CLASS阶段 (JDK的API写到：the retention policy defaults to RetentionPolicy.CLASS.)**

### @Target元注解

@Target元注解决定了一个注解可以标识到哪些成分上，如标识在在类身上，或者属性身上，或者方法身上等成分，@Target默认值为任何元素(成分)

例如：

|  |
| --- |
| **@Target(value={TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL\_VARIABLE})**  @Retention(value=SOURCE)  public @interface SuppressWarnings |

### 其他注解

#### @Deprecated

#### @Override

#### @SuppressWarnings

### 自定义注解

|  |
| --- |
| **package** org.xzh.springboot.annotation;  **public** **enum** EumTrafficLamp {  ***RED***,//红  ***YELLOW***,//黄  ***GREEN***//绿  } |

|  |
| --- |
| **package** org.xzh.springboot.annotation;  /\*\*  \* 元注解  \*  \* **@author** hong  \*  \*/  **public** **@interface** MetaAnnotation {  String value();  } |

|  |
| --- |
| package org.xzh.springboot.annotation;  import java.lang.annotation.ElementType;  import java.lang.annotation.Retention;  import java.lang.annotation.RetentionPolicy;  import java.lang.annotation.Target;  @Retention(RetentionPolicy.RUNTIME)  @Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})  public @interface MyAnnotation {  //为属性指定缺省值  String color() default "blue";    /\*\*  \* 为注解添加value属性，这个value属性很特殊，如果一个注解中只有一个value属性要设置，  \* 那么在设置注解的属性值时，可以省略属性名和等号不写， 直接写属性值，如@SuppressWarnings("deprecation")，  \* 这里的MyAnnotation注解设置了两个String类型的属性，color和value，  \* 因为color属性指定有缺省值，value属性又是属于特殊的属性，因此使用MyAnnotation注解时  \* 可以这样使用MyAnnotation注解："@MyAnnotation(color="red",value="xdp")"  \* 也可以这样使用："@MyAnnotation("孤傲苍狼")"，这样写就表示MyAnnotation注解只有一个value属性要设置，color属性采用缺省值  \* 当一个注解只有一个value属性要设置时，是可以省略"value="的  \*/  String value();    //添加一个int类型数组的属性  int[] arrint() default {1, 2, 3};    //添加一个枚举类型的属性，并指定枚举属性的缺省值，缺省值只能从枚举类EumTrafficLamp中定义的枚举对象中取出任意一个作为缺省值  EumTrafficLamp lamp() default EumTrafficLamp.RED;    //为注解添加一个注解类型的属性,并指定注解属性的缺省值  MetaAnnotation annotationAttr() default @MetaAnnotation("xdp");  } |

|  |
| --- |
| **package** org.xzh.springboot.annotation;  **import** java.lang.reflect.Field;  **import** java.lang.reflect.Method;  **import** org.junit.Test;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  @MyAnnotation(color="grep", value="123", arrint={4,5,6},  lamp=EumTrafficLamp.***GREEN***, annotationAttr=@MetaAnnotation("ddd"))  **public** **class** MyAnnotationTest {    **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(MyAnnotationTest.**class**);  /\*\*  \* 类注解  \*/  @Test  **public** **void** classTest() {  //这里是检查Annotation类是否有注解，这里需要使用反射才能完成对Annotation类的检查  **if**(MyAnnotationTest.**class**.isAnnotationPresent(MyAnnotation.**class**)){  /\*\*  \* 用反射方式获得注解对应的实例对象后，在通过该对象调用属性对应的方法  \* MyAnnotation是一个类，这个类的实例对象annotation是通过反射得到的，这个实例对象是如何创建的呢？  \* 一旦在某个类上使用了@MyAnnotation，那么这个MyAnnotation类的实例对象annotation就会被创建出来了  \*/  MyAnnotation myAnnotation = MyAnnotationTest.**class**.getAnnotation(MyAnnotation.**class**);    ***logger***.info("color:{}", myAnnotation.color());  ***logger***.info("value:{}", myAnnotation.value());  ***logger***.info("arrint:{}", myAnnotation.arrint());  ***logger***.info("lamp:{}", myAnnotation.lamp());  ***logger***.info("annotationAttr:{}", myAnnotation.annotationAttr().value());  }  }    @MyAnnotation("run method")  **public** **void** run(){  ***logger***.info("this method is running...");  }    /\*\*  \* 方法注解  \*  \* **@throws** NoSuchMethodException  \* **@throws** SecurityException  \*/  @Test  **public** **void** methodTest() **throws** NoSuchMethodException, SecurityException{  //利用class获取method  Method run = MyAnnotationTest.**class**.getMethod("run");    //判断方式是否有MyAnnotation注解  **if**(run.isAnnotationPresent(MyAnnotation.**class**)){  MyAnnotation myAnnotation = run.getAnnotation(MyAnnotation.**class**);  ***logger***.info("color:{}", myAnnotation.color());  ***logger***.info("value:{}", myAnnotation.value());  ***logger***.info("arrint:{}", myAnnotation.arrint());  ***logger***.info("lamp:{}", myAnnotation.lamp());  ***logger***.info("annotationAttr:{}", myAnnotation.annotationAttr().value());  }  }    @MyAnnotation("xiezhonghong")  **private** String name;    /\*\*  \* 属性注解  \* **@throws** ClassNotFoundException  \*/  @Test  **public** **void** fieldTest() **throws** ClassNotFoundException{  //获取class  Class<?> clazz = Class.*forName*("org.xzh.springboot.annotation.MyAnnotationTest");  //获取当前类的属性  Field[] fields = clazz.getDeclaredFields();  //取消安全性检查  Field.*setAccessible*(fields, **true**);    **for**(Field field : fields){  **if**(field.isAnnotationPresent(MyAnnotation.**class**)){  MyAnnotation myAnnotation = field.getAnnotation(MyAnnotation.**class**);    ***logger***.info("color:{}", myAnnotation.color());  ***logger***.info("value:{}", myAnnotation.value());  ***logger***.info("arrint:{}", myAnnotation.arrint());  ***logger***.info("lamp:{}", myAnnotation.lamp());  ***logger***.info("annotationAttr:{}", myAnnotation.annotationAttr().value());  }  }  }  } |

## Java常见命令

**1、javac：把java原文件编译成class文件**

用法: javac <options> <source files>

其中, 可能的选项包括:

-g 生成所有调试信息

-g:none 不生成任何调试信息

-g:{lines,vars,source} 只生成某些调试信息

-nowarn 不生成任何警告

-verbose 输出有关编译器正在执行的操作的消息

-deprecation 输出使用已过时的 API 的源位置

-classpath <路径> 指定查找用户类文件和注释处理程序的位置

-cp <路径> 指定查找用户类文件和注释处理程序的位置

-sourcepath <路径> 指定查找输入源文件的位置

-bootclasspath <路径> 覆盖引导类文件的位置

-extdirs <目录> 覆盖所安装扩展的位置

-endorseddirs <目录> 覆盖签名的标准路径的位置

-proc:{none,only} 控制是否执行注释处理和/或编译。

-processor <class1>[,<class2>,<class3>...] 要运行的注释处理程序的名称; 绕过默认的搜索进程

-processorpath <路径> 指定查找注释处理程序的位置

-parameters 生成元数据以用于方法参数的反射

-d <目录> 指定放置生成的类文件的位置

-s <目录> 指定放置生成的源文件的位置

-h <目录> 指定放置生成的本机标头文件的位置

-implicit:{none,class} 指定是否为隐式引用文件生成类文件

-encoding <编码> 指定源文件使用的字符编码

-source <发行版> 提供与指定发行版的源兼容性

-target <发行版> 生成特定 VM 版本的类文件

-profile <配置文件> 请确保使用的 API 在指定的配置文件中可用

-version 版本信息

-help 输出标准选项的提要

-A关键字[=值] 传递给注释处理程序的选项

-X 输出非标准选项的提要

-J<标记> 直接将 <标记> 传递给运行时系统

-Werror 出现警告时终止编译

@<文件名> 从文件读取选项和文件名

**2、javap：把javac编译后的class文件，反编译成java文件**

用法: javap <options> <classes>

其中, 可能的选项包括:

-help --help -? 输出此用法消息

-version 版本信息

-v -verbose 输出附加信息

-l 输出行号和本地变量表

-public 仅显示公共类和成员

-protected 显示受保护的/公共类和成员

-package 显示程序包/受保护的/公共类

和成员 (默认)

-p -private 显示所有类和成员

-c 对代码进行反汇编

-s 输出内部类型签名

-sysinfo 显示正在处理的类的

系统信息 (路径, 大小, 日期, MD5 散列)

-constants 显示最终常量

-classpath <path> 指定查找用户类文件的位置

-cp <path> 指定查找用户类文件的位置

-bootclasspath <path> 覆盖引导类文件的位置

**3、java：执行class文件**

用法: java [-options] class [args...]

(执行类)

或 java [-options] -jar jarfile [args...]

(执行 jar 文件)

其中选项包括:

-d32 使用 32 位数据模型 (如果可用)

-d64 使用 64 位数据模型 (如果可用)

-server 选择 "server" VM

-hotspot 是 "server" VM 的同义词 [已过时]

默认 VM 是 server.

-cp <目录和 zip/jar 文件的类搜索路径>

-classpath <目录和 zip/jar 文件的类搜索路径>

用 ; 分隔的目录, JAR 档案

和 ZIP 档案列表, 用于搜索类文件。

-D<名称>=<值>

设置系统属性

-verbose:[class|gc|jni]

启用详细输出

-version 输出产品版本并退出

-version:<值>

需要指定的版本才能运行

-showversion 输出产品版本并继续

-jre-restrict-search | -no-jre-restrict-search

在版本搜索中包括/排除用户专用 JRE

-? -help 输出此帮助消息

-X 输出非标准选项的帮助

-ea[:<packagename>...|:<classname>]

-enableassertions[:<packagename>...|:<classname>]

按指定的粒度启用断言

-da[:<packagename>...|:<classname>]

-disableassertions[:<packagename>...|:<classname>]

禁用具有指定粒度的断言

-esa | -enablesystemassertions

启用系统断言

-dsa | -disablesystemassertions

禁用系统断言

-agentlib:<libname>[=<选项>]

加载本机代理库 <libname>, 例如 -agentlib:hprof

另请参阅 -agentlib:jdwp=help 和 -agentlib:hprof=help

-agentpath:<pathname>[=<选项>]

按完整路径名加载本机代理库

-javaagent:<jarpath>[=<选项>]

加载 Java 编程语言代理, 请参阅 java.lang.instrument

-splash:<imagepath>

使用指定的图像显示启动屏幕

有关详细信息, 请参阅 http://www.oracle.com/technetwork/java/javase/documentation/index.html。

**4、javah：把class文件生成c/c++的.h头文件**

用法:

javah [options] <classes>

其中, [options] 包括:

-o <file> 输出文件 (只能使用 -d 或 -o 之一)

-d <dir> 输出目录

-v -verbose 启用详细输出

-h --help -? 输出此消息

-version 输出版本信息

-jni 生成 JNI 样式的标头文件 (默认值)

-force 始终写入输出文件

-classpath <path> 从中加载类的路径

-cp <path> 从中加载类的路径

-bootclasspath <path> 从中加载引导类的路径

<classes> 是使用其全限定名称指定的

(例如, java.lang.Object)。

**5、javadoc：生成java的doc文档**

用法: javadoc [options] [packagenames] [sourcefiles] [@files]

-overview <file> 从 HTML 文件读取概览文档

-public 仅显示 public 类和成员

-protected 显示 protected/public 类和成员 (默认值)

-package 显示 package/protected/public 类和成员

-private 显示所有类和成员

-help 显示命令行选项并退出

-doclet <class> 通过替代 doclet 生成输出

-docletpath <path> 指定查找 doclet 类文件的位置

-sourcepath <pathlist> 指定查找源文件的位置

-classpath <pathlist> 指定查找用户类文件的位置

-cp <pathlist> 指定查找用户类文件的位置

-exclude <pkglist> 指定要排除的程序包列表

-subpackages <subpkglist> 指定要递归加载的子程序包

-breakiterator 计算带有 BreakIterator 的第一个语句

-bootclasspath <pathlist> 覆盖由引导类加载器所加载的

类文件的位置

-source <release> 提供与指定发行版的源兼容性

-extdirs <dirlist> 覆盖所安装扩展的位置

-verbose 输出有关 Javadoc 正在执行的操作的信息

-locale <name> 要使用的区域设置, 例如 en\_US 或 en\_US\_WIN

-encoding <name> 源文件编码名称

-quiet 不显示状态消息

-J<flag> 直接将 <flag> 传递到运行时系统

-X 输出非标准选项的提要

通过标准 doclet 提供:

-d <directory> 输出文件的目标目录

-use 创建类和程序包用法页面

-version 包含 @version 段

-author 包含 @author 段

-docfilessubdirs 递归复制文档文件子目录

-splitindex 将索引分为每个字母对应一个文件

-windowtitle <text> 文档的浏览器窗口标题

-doctitle <html-code> 包含概览页面的标题

-header <html-code> 包含每个页面的页眉文本

-footer <html-code> 包含每个页面的页脚文本

-top <html-code> 包含每个页面的顶部文本

-bottom <html-code> 包含每个页面的底部文本

-link <url> 创建指向位于 <url> 的 javadoc 输出的链接

-linkoffline <url> <url2> 利用位于 <url2> 的程序包列表链接至位于 <url> 的文档

-excludedocfilessubdir <name1>:.. 排除具有给定名称的所有文档文件子目录。

-group <name> <p1>:<p2>.. 在概览页面中, 将指定的程序包分组

-nocomment 不生成说明和标记, 只生成声明。

-nodeprecated 不包含 @deprecated 信息

-noqualifier <name1>:<name2>:... 输出中不包括指定限定符的列表。

-nosince 不包含 @since 信息

-notimestamp 不包含隐藏时间戳

-nodeprecatedlist 不生成已过时的列表

-notree 不生成类分层结构

-noindex 不生成索引

-nohelp 不生成帮助链接

-nonavbar 不生成导航栏

-serialwarn 生成有关 @serial 标记的警告

-tag <name>:<locations>:<header> 指定单个参数定制标记

-taglet 要注册的 Taglet 的全限定名称

-tagletpath Taglet 的路径

-charset <charset> 用于跨平台查看生成的文档的字符集。

-helpfile <file> 包含帮助链接所链接到的文件

-linksource 以 HTML 格式生成源文件

-sourcetab <tab length> 指定源中每个制表符占据的空格数

-keywords 使程序包, 类和成员信息附带 HTML 元标记

-stylesheetfile <path> 用于更改生成文档的样式的文件

-docencoding <name> 指定输出的字符编码

java.exe用于启动window console  控制台程序

javaw.exe用于启动 GUI程序

javaws.exe用于web程序。

**6、jcmd：**

Usage: jcmd <pid | main class> <command ...|PerfCounter.print|-f file>

or: jcmd -l

or: jcmd -h

command must be a valid jcmd command for the selected jvm.

Use the command "help" to see which commands are available.

If the pid is 0, commands will be sent to all Java processes.

The main class argument will be used to match (either partially

or fully) the class used to start Java.

If no options are given, lists Java processes (same as -p).

PerfCounter.print display the counters exposed by this process

-f read and execute commands from the file

-l list JVM processes on the local machine

-h this help

6.1：查看 当前机器上所有的 jvm 进程信息

jps、jcmd、jcmd -l

三个命令查到效果是一样

6.2：查看性能统计

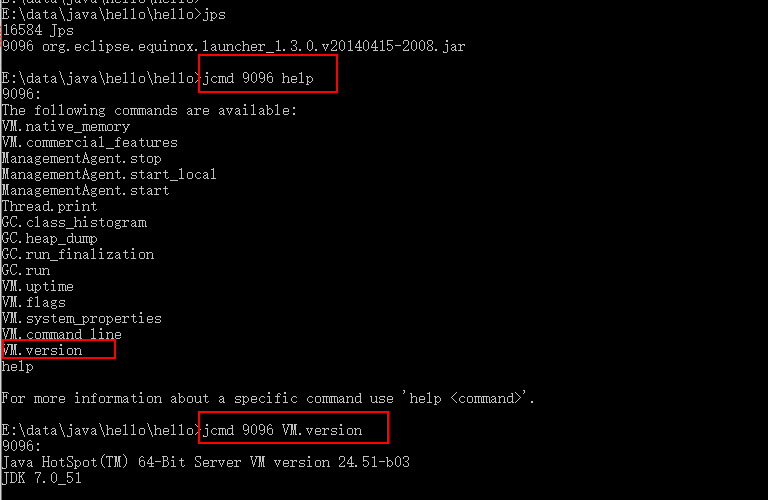
命令：jcmd pid PerfCounter.print

描述：查看指定进程的性能统计信息。



6.3：列出当前运行的java进程可以执行的操作

命令：jcmd PID help



**7、jconsole：jdk自带性能监控工具，通过命令jconsole或者jdk安装bin/jconsole.exe启动**

用法: jconsole [ -interval=n ] [ -notile ] [ -pluginpath <path> ] [ -version ] [ connection ... ]

-interval 将更新间隔设置为 n 秒 (默认值为 4 秒)

-notile 初始不平铺窗口 (对于两个或多个连接)

-pluginpath 指定 jconsole 用于查找插件的路径

-version 输出程序版本

connection = pid || host:port || JMX URL (service:jmx:<协议>://...)

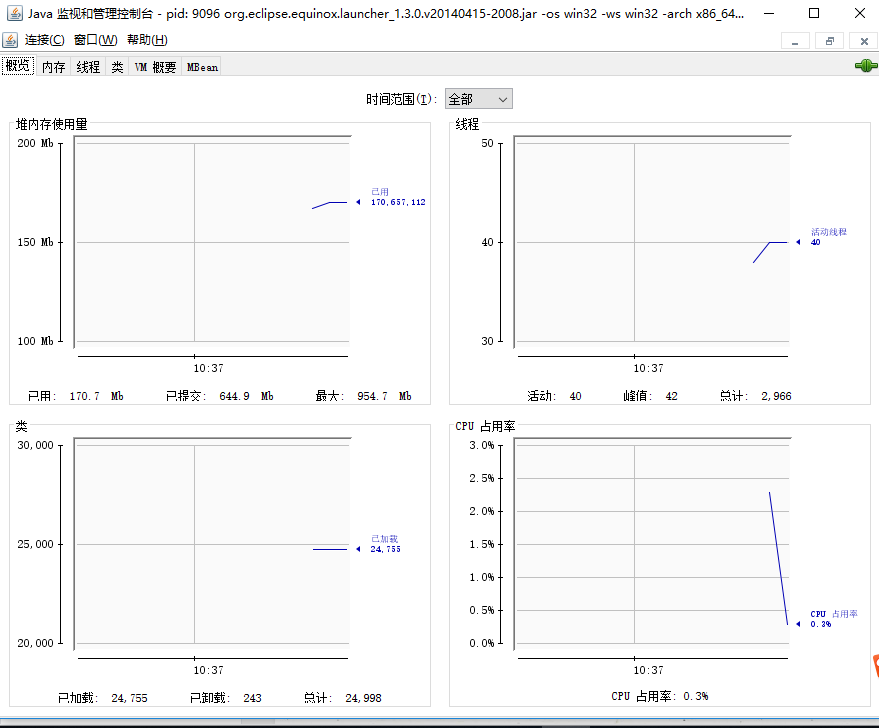
pid 目标进程的进程 ID

host 远程主机名或 IP 地址

port 远程连接的端口号

-J 指定运行 jconsole 的 Java 虚拟机

的输入参数



**8.jdb：使用jdb命令调试程序**

用法: jdb <options> <class> <arguments>

其中, 选项包括:

-help 输出此消息并退出

-sourcepath <由 ";" 分隔的目录>

要在其中查找源文件的目录

-attach <address>

使用标准连接器附加到指定地址处正在运行的 VM

-listen <address>

等待正在运行的 VM 使用标准连接器在指定地址处连接

-listenany

等待正在运行的 VM 使用标准连接器在任何可用地址处连接

-launch

立即启动 VM 而不是等待 'run' 命令

-listconnectors 列出此 VM 中的可用连接器

-connect <connector-name>:<name1>=<value1>,...

使用所列参数值通过指定的连接器连接到目标 VM

-dbgtrace [flags] 输出信息供调试jdb

-tclient 在 HotSpot(TM) 客户机编译器中运行应用程序

-tserver 在 HotSpot(TM) 服务器编译器中运行应用程序

转发到被调试进程的选项:

-v -verbose[:class|gc|jni]

启用详细模式

-D<name>=<value> 设置系统属性

-classpath <由 ";" 分隔的目录>

列出要在其中查找类的目录

-X<option> 非标准目标 VM 选项

<class> 是要开始调试的类的名称

<arguments> 是传递到 <class> 的 main() 方法的参数

要获得命令的帮助, 请在jdb提示下键入 'help'

**9、jdeps：java8才引入的命令，类依赖分析器**

用法: jdeps <options> <classes...>

其中 <classes> 可以是 .class 文件, 目录, JAR 文件的路径名,

也可以是全限定类名。可能的选项包括:

-dotoutput <dir> DOT 文件输出的目标目录

-s -summary 仅输出被依赖对象概要

-v -verbose 输出所有类级别被依赖对象

等同于 -verbose:class -filter:none。

-verbose:package 默认情况下输出程序包级别被依赖对象,

不包括同一程序包中的被依赖对象

-verbose:class 默认情况下输出类级别被依赖对象,

不包括同一程序包中的被依赖对象

-cp <path> -classpath <path> 指定查找类文件的位置

-p <pkgname> -package <pkgname> 查找与给定程序包名称匹配的被依赖对象

(可多次指定)

-e <regex> -regex <regex> 查找与指定模式匹配的被依赖对象

(-p 和 -e 互相排斥)

-f <regex> -filter <regex> 筛选与指定模式匹配的被依赖对象

如果多次指定, 则将使用最后一个被依赖对象。

-filter:package 筛选位于同一程序包内的被依赖对象 (默认)

-filter:archive 筛选位于同一档案内的被依赖对象

-filter:none 不使用 -filter:package 和 -filter:archive 筛选

通过 -filter 选项指定的筛选仍旧适用。

-include <regex> 将分析限制为与模式匹配的类

此选项筛选要分析的类的列表。

它可以与向被依赖对象应用模式的

-p 和 -e 结合使用

-P -profile 显示配置文件或包含程序包的文件

-apionly 通过公共类 (包括字段类型, 方法参数

类型, 返回类型, 受控异常错误类型

等) 的公共和受保护成员的签名

限制对 API (即被依赖对象)

进行分析

-R -recursive 递归遍历所有被依赖对象。

-R 选项表示 -filter:none。如果指定了 -p, -e, -f

选项, 则只分析匹配的

被依赖对象。

-jdkinternals 在 JDK 内部 API 上查找类级别的被依赖对象。

默认情况下, 它分析 -classpath 上的所有类

和输入文件, 除非指定了 -include 选项。

此选项不能与 -p, -e 和 -s 选项一起使用。

警告: 在下一个发行版中可能无法访问

JDK 内部 API。

-version 版本信息



**10、jhat：分析Java堆。该jhat命令解析Java堆转储文件并启动Web服务器。jhat命令可以让你浏览堆转储。支持OQL语法。**

**使用*jmap*等方法生产堆文件之后，使用其进行分析。**

Usage: jhat [-stack <bool>] [-refs <bool>] [-port <port>] [-baseline <file>] [-debug <int>] [-version] [-h|-help] <file>

-J<flag> Pass <flag> directly to the runtime system. For

example, -J-mx512m to use a maximum heap size of 512MB

-stack false: Turn off tracking object allocation call stack.

-refs false: Turn off tracking of references to objects

-port <port>: Set the port for the HTTP server. Defaults to 7000

-exclude <file>: Specify a file that lists data members that should

be excluded from the reachableFrom query.

-baseline <file>: Specify a baseline object dump. Objects in

both heap dumps with the same ID and same class will

be marked as not being "new".

-debug <int>: Set debug level.

0: No debug output

1: Debug hprof file parsing

2: Debug hprof file parsing, no server

-version Report version number

-h|-help Print this help and exit

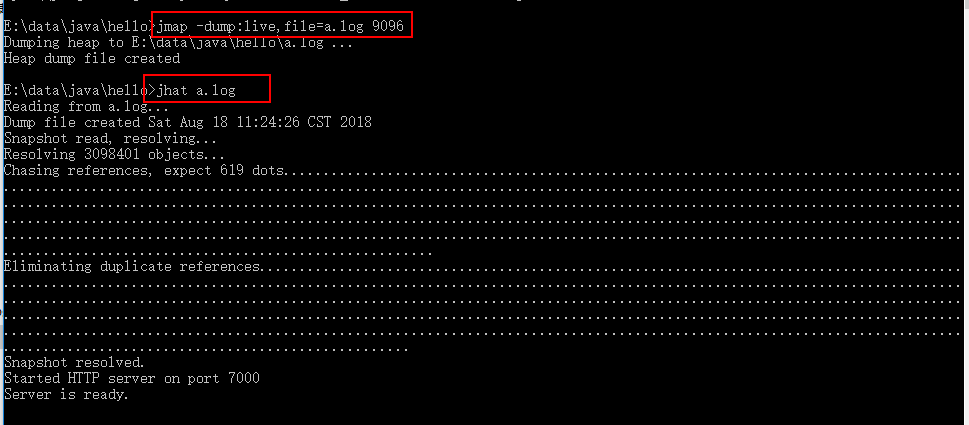
<file> The file to read

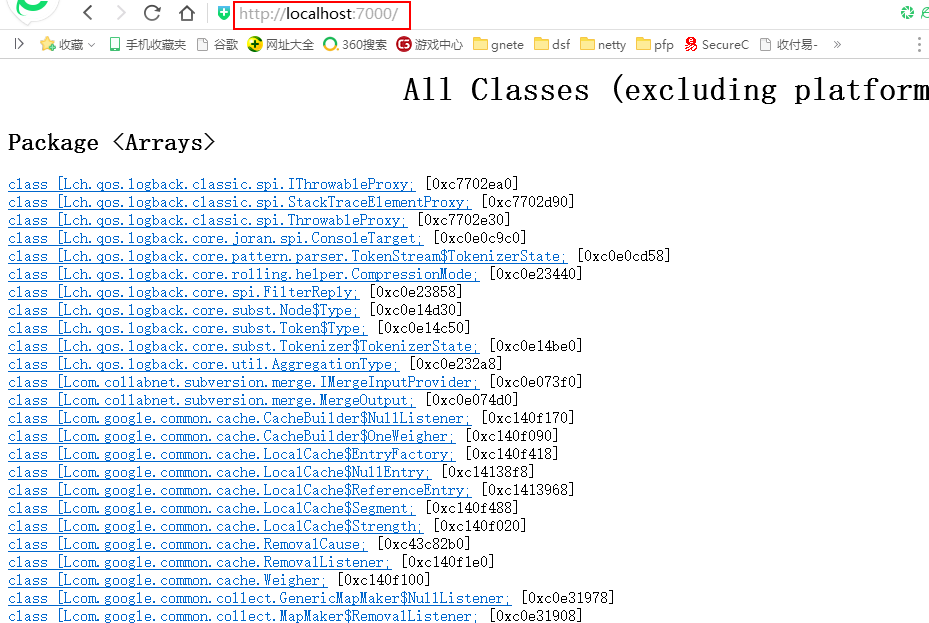
For a dump file that contains multiple heap dumps,

you may specify which dump in the file

by appending "#<number>" to the file name, i.e. "foo.hprof#3".

All boolean options default to "true"





**11、jmap：堆分析工具，统计堆对象数量，也可以用来生成jvm堆文件(heap dump文件)。**

Usage:

jmap [option] <pid>

(to connect to running process)

jmap [option] <executable <core>

(to connect to a core file)

jmap [option] [server\_id@]<remote server IP or hostname>

(to connect to remote debug server)

where <option> is one of:

<none> to print same info as Solaris pmap

-heap to print java heap summary

-histo[:live] to print histogram of java object heap; if the "live"

suboption is specified, only count live objects

-clstats to print class loader statistics

-finalizerinfo to print information on objects awaiting finalization

-dump:<dump-options> to dump java heap in hprof binary format

dump-options:

live dump only live objects; if not specified,

all objects in the heap are dumped.

format=b binary format

file=<file> dump heap to <file>

Example: jmap -dump:live,format=b,file=heap.bin <pid>

-F force. Use with -dump:<dump-options> <pid> or -histo

to force a heap dump or histogram when <pid> does not

respond. The "live" suboption is not supported

in this mode.

-h | -help to print this help message

-J<flag> to pass <flag> directly to the runtime system

**12、jinfo：jinfo是jdk自带的命令，可以用来查看正在运行的Java应用程序的扩展参数，甚至支持在运行时，修改部分参数。*实时地查看和调整虚拟机各项参数***

Usage:

jinfo [option] <pid>

(to connect to running process)

jinfo [option] <executable <core>

(to connect to a core file)

jinfo [option] [server\_id@]<remote server IP or hostname>

(to connect to remote debug server)

where <option> is one of:

-flag <name> to print the value of the named VM flag

-flag [+|-]<name> to enable or disable the named VM flag

-flag <name>=<value> to set the named VM flag to the given value

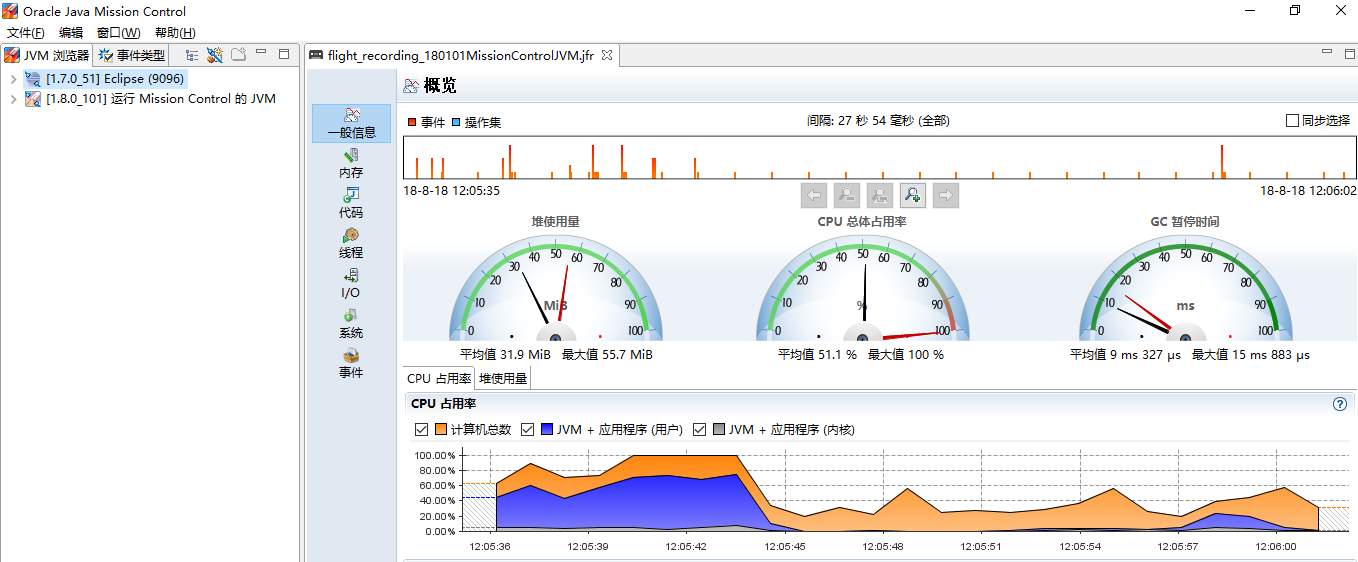
-flags to print VM flags

-sysprops to print Java system properties

<no option> to print both of the above

-h | -help to print this help message

**13、jmc：java mission control，用于本地/远程监控JVM的运行状态的管理工具**



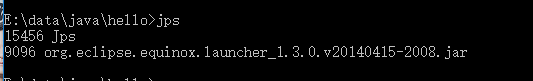
**14、jps：查看java进程**

usage: jps [-help]

jps [-q] [-mlvV] [<hostid>]

Definitions:

<hostid>: <hostname>[:<port>]



**15、jrunscript：支持java与脚本相互调用，就是在jvm执行脚本（如JavaScript）**

Usage: jrunscript [options] [arguments...]

where [options] include:

-classpath <path> Specify where to find user class files

-cp <path> Specify where to find user class files

-D<name>=<value> Set a system property

-J<flag> Pass <flag> directly to the runtime system

-l <language> Use specified scripting language

-e <script> Evaluate given script

-encoding <encoding> Specify character encoding used by script files

-f <script file> Evaluate given script file

-f - Interactive mode, read script from standard input

If this is used, this should be the last -f option

-help Print this usage message and exit

-? Print this usage message and exit

-q List all scripting engines available and exit

If [arguments..] are present and if no -e or -f option is used, then first

argument is script file and the rest of the arguments, if any, are passed

as script arguments. If [arguments..] and -e or -f option is used, then all

[arguments..] are passed as script arguments. If [arguments..], -e, -f are

missing, then interactive mode is used.

**16：jstack：java虚拟机自带的一种堆栈跟踪工具。jstack命令主要用来查看Java线程的调用堆栈的，可以用来分析线程问题（如死锁）。**

Usage:

jstack [-l] <pid>

(to connect to running process)

jstack -F [-m] [-l] <pid>

(to connect to a hung process)

jstack [-m] [-l] <executable> <core>

(to connect to a core file)

jstack [-m] [-l] [server\_id@]<remote server IP or hostname>

(to connect to a remote debug server)

Options:

-F to force a thread dump. Use when jstack <pid> does not respond (process is hung)

-m to print both java and native frames (mixed mode)

-l long listing. Prints additional information about locks

-h or -help to print this help message

**17：jstat：jstat命令可以查看堆内存各部分的使用量，以及加载类的数量。**

Usage: jstat -help|-options

jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]

Definitions:

<option> An option reported by the -options option

<vmid> Virtual Machine Identifier. A vmid takes the following form:

<lvmid>[@<hostname>[:<port>]]

Where <lvmid> is the local vm identifier for the target

Java virtual machine, typically a process id; <hostname> is

the name of the host running the target Java virtual machine;

and <port> is the port number for the rmiregistry on the

target host. See the jvmstat documentation for a more complete

description of the Virtual Machine Identifier.

<lines> Number of samples between header lines.

<interval> Sampling interval. The following forms are allowed:

<n>["ms"|"s"]

Where <n> is an integer and the suffix specifies the units as

milliseconds("ms") or seconds("s"). The default units are "ms".

<count> Number of samples to take before terminating.

-J<flag> Pass <flag> directly to the runtime system.

-options参数如下

-class

-compiler

-gc

-gccapacity

-gccause

-gcmetacapacity

-gcnew

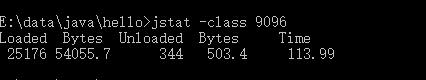
-gcnewcapacity

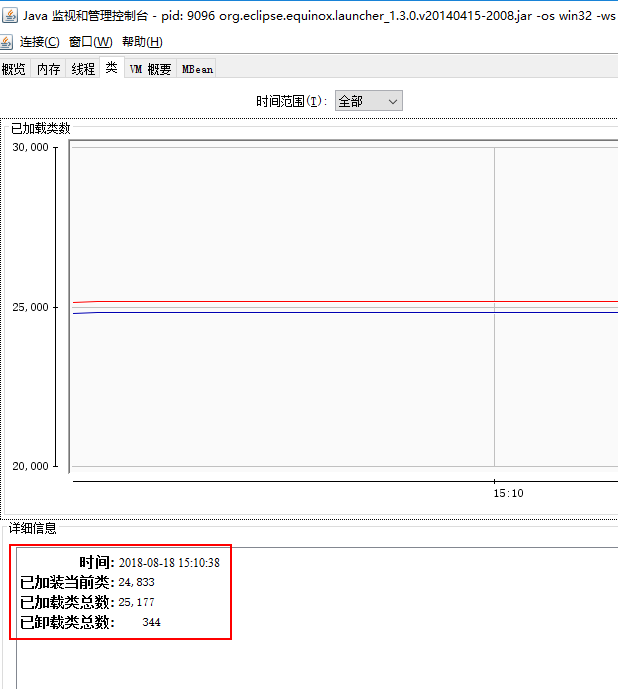
-gcold

-gcoldcapacity

-gcutil

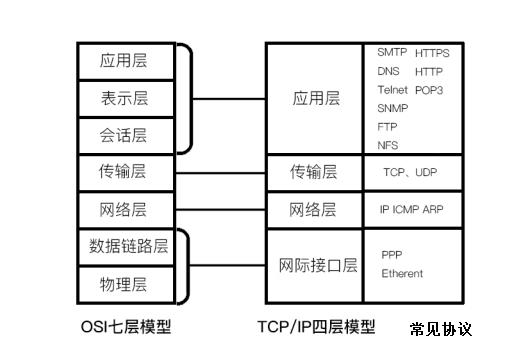
-printcompilation





# 网络知识

## OSI七层网络模型



## TCP/IP协议

TCP/IP不是一个协议，而是一个协议族的统称。里面包括了IP协议，IMCP协议，TCP协议，以及我们更加熟悉的http、ftp、pop3协议等等。

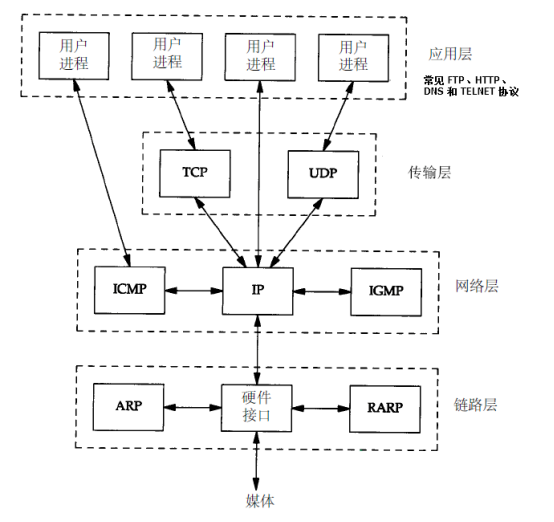
TCP/IP协议分为四层：

(1)应用层：应用程序通过这一层访问网络，常见 FTP、HTTP、DNS 和 TELNET 协议；

(2)传输层：TCP 协议和 UDP 协议；

(3)网络层：IP 协议，ARP、RARP 协议，ICMP 协议等；

(4)网络接口层：是 TCP/IP 协议的基层，负责数据帧的发送和接收。



### 基础知识

**1.IP 地址**

网络上每一个节点都必须有一个独立的 IP 地址，通常使用的 IP 地址是一个 32bit 的数字，被分成 4 组，例如，255.255.255.255 就是一个 IP 地址。IP地址就是计算机网络组成的最小单位。

在 Linux 系统中，可以用 ifconfig -a 命令查看自己的 IP 地址，windows的DOS中可以用ipconfing查看

**2.域名**

用 12 位数字组成的 IP 地址，在实际应用时，用户一般不需要记住 IP 地址，互联网给每个 IP 地址起了一个别名，习惯上称作域名。

可以使用命令 nslookup 或者 ping 在Linux中查看与域名相对应的 IP 地址。

**3.MAC 地址**

MAC（Media Access Control）地址，或称为物理地址、硬件地址，用来定义互联网中设备的位置。

在 TCP/IP 层次模型中，网络层管理 IP 地址，链路层则负责 MAC 地址。因此每个网络位置会有一个专属于它的 IP 地址，而每个主机会有一个专属于它 MAC 地址。

### 交互时数据处理方式

**封装**：当应用程序发送数据的时候，数据在协议层次当中从顶向下通过每一层，每一层都会对数据增加一些首部或尾部信息，这样的信息称之为协议数据单元（Protocol Data Unit，缩写为PDU），在分层协议系统里，在指定的协议层上传送的数据单元，包含了该层的协议控制信息和用户信息。如下图所示：

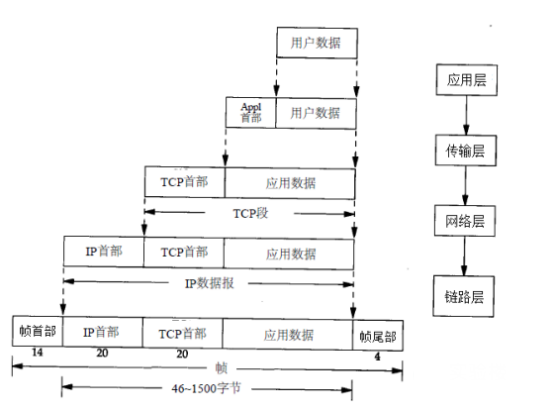
**· 物理层（一层）PDU指数据位（Bit）**

**· 数据链路层（二层）PDU指数据帧（Frame）**

**· 网络层（三层）PDU指数据包（Packet）**

**· 传输层（四层）PDU指数据段（Segment）**

**· 第五层以上为数据（data）**

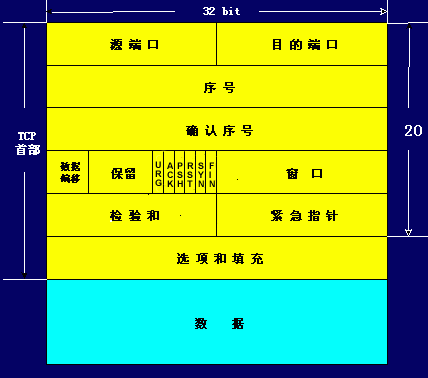


**分用**：当主机收到一个数据帧时，数据就从协议层底向上升，通过每一层时，检查并去掉对应层次的报文首部或尾部，与封装过程正好相反。

## TCP/UDP

### TCP

#### TCP报文格式



16位源端口号：16位的源端口中包含初始化通信的端口。源端口和源IP地址的作用是标识报文的返回地址。

16位目的端口号：16位的目的端口域定义传输的目的。这个端口指明报文接收计算机上的应用程序地址接口。

32位序号：32位的序列号由接收端计算机使用，重新分段的报文成最初形式。当SYN出现，序列码实际上是初始序列码（Initial Sequence Number，ISN），而第一个数据字节是ISN+1。这个序列号（序列码）可用来补偿传输中的不一致。

32位确认序号：32位的序列号由接收端计算机使用，重组分段的报文成最初形式。如果设置了ACK控制位，这个值表示一个准备接收的包的序列码。

4位首部长度：4位包括TCP头大小，指示何处数据开始。

保留（6位）：6位值域，这些位必须是0。为了将来定义新的用途而保留。

标志：6位标志域。表示为：紧急标志、有意义的应答标志、推、重置连接标志、同步序列号标志、完成发送数据标志。按照顺序排列是：URG、ACK、PSH、RST、SYN、FIN。

16位窗口大小：用来表示想收到的每个TCP数据段的大小。TCP的流量控制由连接的每一端通过声明的窗口大小来提供。窗口大小为字节数，起始于确认序号字段指明的值，这个值是接收端正期望接收的字节。窗口大小是一个16字节字段，因而窗口大小最大为65535字节。

16位校验和：16位TCP头。源机器基于数据内容计算一个数值，收信息机要与源机器数值 结果完全一样，从而证明数据的有效性。检验和覆盖了整个的TCP报文段：这是一个强制性的字段，一定是由发送端计算和存储，并由接收端进行验证的。

16位紧急指针：指向后面是优先数据的字节，在URG标志设置了时才有效。如果URG标志没有被设置，紧急域作为填充。加快处理标示为紧急的数据段。

选项：长度不定，但长度必须为1个字节。如果没有选项就表示这个1字节的域等于0。

数据：该TCP协议包负载的数据。

在上述字段中，6位标志域的各个选项功能如下：

URG：紧急标志。紧急标志为"1"表明该位有效。

ACK：确认标志。表明确认编号栏有效。大多数情况下该标志位是置位的。TCP报头内的确认编号栏内包含的确认编号（w+1）为下一个预期的序列编号，同时提示远端系统已经成功接收所有数据。

PSH：推标志。该标志置位时，接收端不将该数据进行队列处理，而是尽可能快地将数据转由应用处理。在处理Telnet或rlogin等交互模式的连接时，该标志总是置位的。

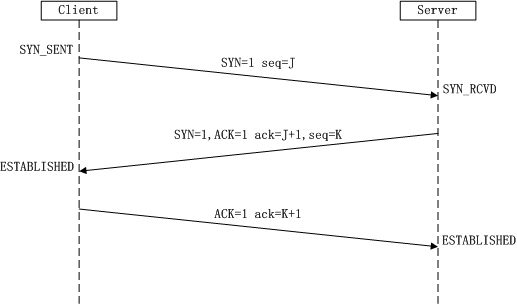
RST：复位标志。用于复位相应的TCP连接。

SYN：同步标志。表明同步序列编号栏有效。该标志仅在三次握手建立TCP连接时有效。它提示TCP连接的服务端检查序列编号，该序列编号为TCP连接初始端（一般是客户端）的初始序列编号。在这里，可以把TCP序列编号看作是一个范围从0到4，294，967，295的32位计数器。通过TCP连接交换的数据中每一个字节都经过序列编号。在TCP报头中的序列编号栏包括了TCP分段中第一个字节的序列编号。

FIN：结束标志。

#### TCP三次握手

所谓三次握手（Three-Way Handshake）即建立TCP连接，就是指建立一个TCP连接时，需要客户端和服务端总共发送3个包以确认连接的建立。在socket编程中，这一过程由客户端执行connect来触发，整个流程如下图所示：



（1）第一次握手：Client将标志位SYN置为1，随机产生一个值seq=J，并将该数据包发送给Server，Client进入SYN\_SENT状态，等待Server确认。

（2）第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack=J+1，随机产生一个值seq=K，并将该数据包发送给Client以确认连接请求，Server进入SYN\_RCVD状态。

（3）第三次握手：Client收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给Server，Server检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

简单来说，就是

1、建立连接时，客户端发送SYN包（SYN=i）到服务器，并进入到SYN-SEND状态，等待服务器确认

2、服务器收到SYN包，必须确认客户的SYN（ack=i+1）,同时自己也发送一个SYN包（SYN=k）,即SYN+ACK包，此时服务器进入SYN-RECV状态

3、客户端收到服务器的SYN+ACK包，向服务器发送确认报ACK（ack=k+1）,此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手，客户端与服务器开始传送数据。

**SYN攻击**：

在三次握手过程中，Server发送SYN-ACK之后，收到Client的ACK之前的TCP连接称为半连接（half-open connect），此时Server处于SYN\_RCVD状态，当收到ACK后，Server转入ESTABLISHED状态。SYN攻击就是Client在短时间内伪造大量不存在的IP地址，并向Server不断地发送SYN包，Server回复确认包，并等待Client的确认，由于源地址是不存在的，因此，Server需要不断重发直至超时，这些伪造的SYN包将产时间占用未连接队列，导致正常的SYN请求因为队列满而被丢弃，从而引起网络堵塞甚至系统瘫痪。SYN攻击时一种典型的DDOS攻击，检测SYN攻击的方式非常简单，即当Server上有大量半连接状态且源IP地址是随机的，则可以断定遭到SYN攻击了，使用如下命令可以让之现行：

#netstat -nap | grep SYN\_RECV

#### TCP四次挥手

所谓四次挥手（Four-Way Wavehand）即终止TCP连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程由客户端或服务端任一方执行close来触发，整个流程如下图所示：



由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭，上图描述的即是如此。

（1）第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态。

（2）第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE\_WAIT状态。

（3）第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST\_ACK状态。

（4）第四次挥手：Client收到FIN后，Client进入TIME\_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。

**为什么建立连接是三次握手，而关闭连接却是四次挥手呢？**

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

**为什么TIME\_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？**

原因有二：

一、保证TCP协议的全双工连接能够可靠关闭

二、保证这次连接的重复数据段从网络中消失

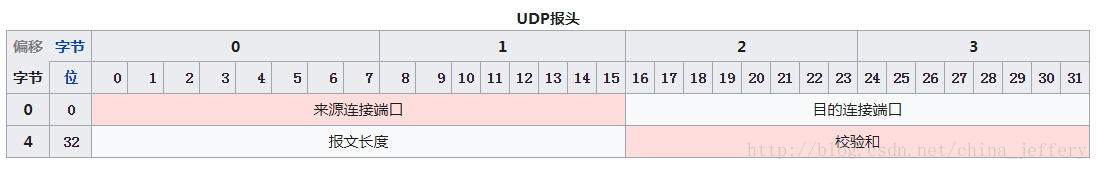
先说第一点，如果Client直接CLOSED了，那么由于IP协议的不可靠性或者是其它网络原因，导致Server没有收到Client最后回复的ACK。那么Server就会在超时之后继续发送FIN，此时由于Client已经CLOSED了，就找不到与重发的FIN对应的连接，最后Server就会收到RST而不是ACK，Server就会以为是连接错误把问题报告给高层。这样的情况虽然不会造成数据丢失，但是却导致TCP协议不符合可靠连接的要求。所以，Client不是直接进入CLOSED，而是要保持TIME\_WAIT，当再次收到FIN的时候，能够保证对方收到ACK，最后正确的关闭连接。

再说第二点，如果Client直接CLOSED，然后又再向Server发起一个新连接，我们不能保证这个新连接与刚关闭的连接的端口号是不同的。也就是说有可能新连接和老连接的端口号是相同的。一般来说不会发生什么问题，但是还是有特殊情况出现：假设新连接和已经关闭的老连接端口号是一样的，如果前一次连接的某些数据仍然滞留在网络中，这些延迟数据在建立新连接之后才到达Server，由于新连接和老连接的端口号是一样的，又因为TCP协议判断不同连接的依据是socket pair，于是，TCP协议就认为那个延迟的数据是属于新连接的，这样就和真正的新连接的数据包发生混淆了。所以TCP连接还要在TIME\_WAIT状态等待2倍MSL，这样可以保证本次连接的所有数据都从网络中消失。

### UDP

UDP是User Datagram Protocol的简称，中文名是用户数据报协议，是OSI参考模型中的传输层协议，它是一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

#### UDP报文格式



UDP报头由4个部分组成，其中两个是可选的（粉红背景标出部分）：

1、各16bit的来源端口和目的端口用来标记发送和接受的应用进程。因为UDP不需要应答，所以来源端口是可选的，如果来源端口不用，那么置为零。

2、在目的端口后面是长度固定的以字节为单位的报文长度域，用来指定UDP数据报包括数据部分的长度，长度最小值为8byte。

3、首部剩下地16bit是用来对首部和数据部分一起做校验和（Checksum）的，这部分是可选的，但在实际应用中一般都使用这一功能。

4、UDP和TCP的校验和都覆盖到了他们的首部和数据，而之前介绍的IP首部的校验和只覆盖了IP首部。

**应用场景：**

由于缺乏可靠性且属于非连接导向协议，UDP的应用一般必须允许一定量的丢包、出错和复制粘贴。但有些应用，比如TFTP，需要可靠性保证，则必须在应用层增加根本的可靠机制。但是绝大多数UDP应用都不需要可靠机制，甚至可能因为引入可靠机制而降低性能。流媒体、即时多媒体游戏和IP电话（VoIP）就是典型的UDP应用。如果某个应用需要很高的可靠性，那么可以用传输控制协议（即TCP协议）来代替UDP。

使用UDP协议的应用有：域名系统（DNS）、简单网络管理协议（SNMP）、动态主机配置协议（DHCP）、路由信息协议（RIP）等等。因为UDP不属于连接型协议，因而具有资源消耗小，处理速度快的优点，所以通常音频、视频和普通数据在传送时使用UDP较多，因为它们即使偶尔丢失几个数据包，也不会对接收结果产生太大影响。

### TCP与UDP区别

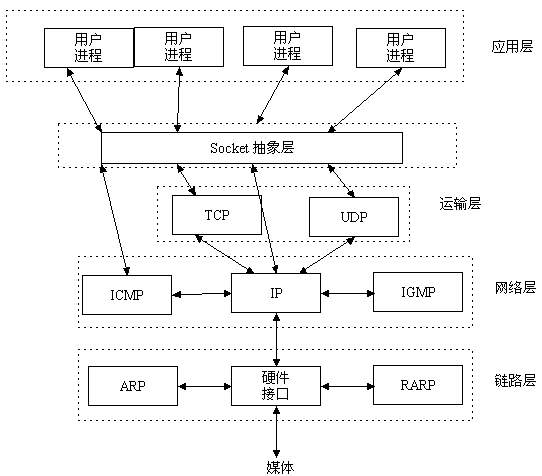
|  |  |  |
| --- | --- | --- |
| 特征点 | TCP | UDP |
| 是否连接 | 面向连接 | 面向非连接 |
| 传输可靠性 | 可靠 | 会丢包、不可靠 |
| 应用场景 | 传输数据量大 | 数据量小 |
| 速度 | 慢 | 块 |

UDP(用户数据报协议)是一个简单的面向数据报的运输层协议。UDP不提供可靠性，它只是把应用程序传给IP层的数据报发送出去，但是并不能保证它们能到达目的地。由于UDP在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快。

由于UDP缺乏拥塞控制（congestion control），需要基于网络的机制来减少因失控和高速UDP流量负荷而导致的拥塞崩溃效应。换句话说，因为UDP发送者不能够检测拥塞，所以像使用包队列和丢弃技术的路由器这样的网络基本设备往往就成为降低UDP过大通信量的有效工具。数据报拥塞控制协议（DCCP）设计成通过在诸如流媒体类型的高速率UDP流中，增加主机拥塞控制，来减小这个潜在的问题。

## Socket

Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。



## Http/Https

### Http

Hyper Text Transfer Protocol，超文本传输协议，是一种建立在TCP上的无状态连接，整个基本的工作流程是客户端发送一个HTTP请求，说明客户端想要访问的资源和请求的动作，服务端收到请求之后，服务端开始处理请求，并根据请求做出相应的动作访问服务器资源，最后通过发送HTTP响应把结果返回给客户端。其中一个请求的开始到一个响应的结束称为事务，当一个事物结束后还会在服务端添加一条日志条目。

#### 请求

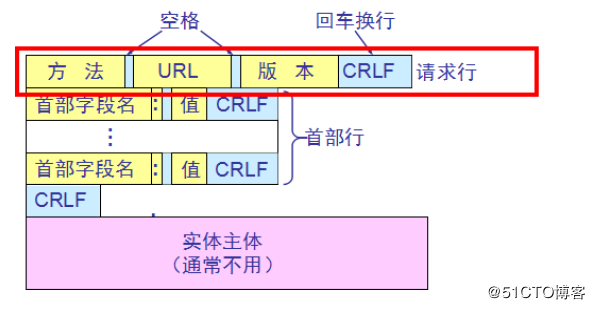
HTTP请求是客户端往服务端发送请求动作，告知服务器自己的要求。

HTTP请求由状态行、请求头、请求正文三部分组成：

状态行：包括请求方式Method、资源路径URL、协议版本Version；

请求头：包括一些访问的域名、用户代理、Cookie等信息；

请求正文：就是HTTP请求的数据。



#### 响应

在接收和解释请求消息后，服务器返回一个HTTP响应消息。

HTTP响应也是由三个部分组成，分别是：状态行、消息报头、响应正文  
1、状态行格式如下：  
HTTP-Version Status-Code Reason-Phrase CRLF  
其中，HTTP-Version表示服务器HTTP协议的版本；Status-Code表示服务器发回的响应状态代码；Reason-Phrase表示状态代码的文本描述。  
状态代码有三位数字组成，第一个数字定义了响应的类别，且有五种可能取值：  
1xx：指示信息--表示请求已接收，继续处理  
2xx：成功--表示请求已被成功接收、理解、接受  
3xx：重定向--要完成请求必须进行更进一步的操作  
4xx：客户端错误--请求有语法错误或请求无法实现  
5xx：服务器端错误--服务器未能实现合法的请求  
常见状态代码、状态描述、说明：  
200 OK      //客户端请求成功  
400 Bad Request  //客户端请求有语法错误，不能被服务器所理解  
401 Unauthorized //请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用   
403 Forbidden  //服务器收到请求，但是拒绝提供服务  
404 Not Found  //请求资源不存在，eg：输入了错误的URL  
500 Internal Server Error //服务器发生不可预期的错误  
503 Server Unavailable  //服务器当前不能处理客户端的请求，一段时间后可能恢复正常  
eg：HTTP/1.1 200 OK （CRLF）

2、响应报头后述

3、响应正文就是服务器返回的资源的内容

#### 报文头

HTTP消息由客户端到服务器的请求和服务器到客户端的响应组成。请求消息和响应消息都是由开始行（对于请求消息，开始行就是请求行，对于响应消息，开始行就是状态行），消息报头（可选），空行（只有CRLF的行），消息正文（可选）组成。

HTTP消息报头包括普通报头、请求报头、响应报头、实体报头。  
每一个报头域都是由名字+“：”+空格+值 组成，消息报头域的名字是大小写无关的。

1、普通报头  
在普通报头中，有少数报头域用于所有的请求和响应消息，但并不用于被传输的实体，只用于传输的消息。  
eg：  
Cache-Control   用于指定缓存指令，缓存指令是单向的（响应中出现的缓存指令在请求中未必会出现），且是独立的（一个消息的缓存指令不会影响另一个消息处理的缓存机制），HTTP1.0使用的类似的报头域为Pragma。  
请求时的缓存指令包括：no-cache（用于指示请求或响应消息不能缓存）、no-store、max-age、max-stale、min-fresh、only-if-cached;  
响应时的缓存指令包括：public、private、no-cache、no-store、no-transform、must-revalidate、proxy-revalidate、max-age、s-maxage.  
eg：为了指示IE浏览器（客户端）不要缓存页面，服务器端的JSP程序可以编写如下：response.sehHeader("Cache-Control","no-cache");  
//response.setHeader("Pragma","no-cache");作用相当于上述代码，通常两者//合用  
这句代码将在发送的响应消息中设置普通报头域：Cache-Control:no-cache

Date普通报头域表示消息产生的日期和时间

Connection普通报头域允许发送指定连接的选项。例如指定连接是连续，或者指定“close”选项，通知服务器，在响应完成后，关闭连接

2、请求报头  
请求报头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息。  
常用的请求报头  
Accept  
Accept请求报头域用于指定客户端接受哪些类型的信息。eg：Accept：image/gif，表明客户端希望接受GIF图象格式的资源；Accept：text/html，表明客户端希望接受html文本。  
Accept-Charset  
Accept-Charset请求报头域用于指定客户端接受的字符集。eg：Accept-Charset:iso-8859-1,gb2312.如果在请求消息中没有设置这个域，缺省是任何字符集都可以接受。  
Accept-Encoding  
Accept-Encoding请求报头域类似于Accept，但是它是用于指定可接受的内容编码。eg：Accept-Encoding:gzip.deflate.如果请求消息中没有设置这个域服务器假定客户端对各种内容编码都可以接受。  
Accept-Language  
Accept-Language请求报头域类似于Accept，但是它是用于指定一种自然语言。eg：Accept-Language:zh-cn.如果请求消息中没有设置这个报头域，服务器假定客户端对各种语言都可以接受。  
Authorization  
Authorization请求报头域主要用于证明客户端有权查看某个资源。当浏览器访问一个页面时，如果收到服务器的响应代码为401（未授权），可以发送一个包含Authorization请求报头域的请求，要求服务器对其进行验证。  
Host（发送请求时，该报头域是必需的）  
Host请求报头域主要用于指定被请求资源的Internet主机和端口号，它通常从HTTP URL中提取出来的，eg：  
我们在浏览器中输入：<http://www.guet.edu.cn/index.html>  
浏览器发送的请求消息中，就会包含Host请求报头域，如下：  
Host：[www.guet.edu.cn](http://www.guet.edu.cn/)  
此处使用缺省端口号80，若指定了端口号，则变成：Host：[www.guet.edu.cn](http://www.guet.edu.cn/):指定端口号  
User-Agent  
我们上网登陆论坛的时候，往往会看到一些欢迎信息，其中列出了你的操作系统的名称和版本，你所使用的浏览器的名称和版本，这往往让很多人感到很神奇，实际 上，服务器应用程序就是从User-Agent这个请求报头域中获取到这些信息。User-Agent请求报头域允许客户端将它的操作系统、浏览器和其它 属性告诉服务器。不过，这个报头域不是必需的，如果我们自己编写一个浏览器，不使用User-Agent请求报头域，那么服务器端就无法得知我们的信息 了。  
请求报头举例：  
GET /form.html HTTP/1.1 (CRLF)  
Accept:image/gif,image/x-xbitmap,image/jpeg,application/x-shockwave-flash,application/vnd.ms-excel,application/vnd.ms-powerpoint,application/msword,\*/\* (CRLF)  
Accept-Language:zh-cn (CRLF)  
Accept-Encoding:gzip,deflate (CRLF)  
If-Modified-Since:Wed,05 Jan 2007 11:21:25 GMT (CRLF)  
If-None-Match:W/"80b1a4c018f3c41:8317" (CRLF)  
User-Agent:Mozilla/4.0(compatible;MSIE6.0;Windows NT 5.0) (CRLF)  
Host:www.guet.edu.cn (CRLF)  
Connection:Keep-Alive (CRLF)  
(CRLF)

3、响应报头  
响应报头允许服务器传递不能放在状态行中的附加响应信息，以及关于服务器的信息和对Request-URI所标识的资源进行下一步访问的信息。  
常用的响应报头  
Location  
Location响应报头域用于重定向接受者到一个新的位置。Location响应报头域常用在更换域名的时候。  
Server  
Server响应报头域包含了服务器用来处理请求的软件信息。与User-Agent请求报头域是相对应的。下面是  
Server响应报头域的一个例子：  
Server：Apache-Coyote/1.1  
WWW-Authenticate  
WWW-Authenticate响应报头域必须被包含在401（未授权的）响应消息中，客户端收到401响应消息时候，并发送Authorization报头域请求服务器对其进行验证时，服务端响应报头就包含该报头域。  
eg：WWW-Authenticate:Basic realm="Basic Auth Test!"  //可以看出服务器对请求资源采用的是基本验证机制。

4、实体报头  
请求和响应消息都可以传送一个实体。一个实体由实体报头域和实体正文组成，但并不是说实体报头域和实体正文要在一起发送，可以只发送实体报头域。实体报头定义了关于实体正文（eg：有无实体正文）和请求所标识的资源的元信息。  
常用的实体报头  
Content-Encoding  
Content-Encoding实体报头域被用作媒体类型的修饰符，它的值指示了已经被应用到实体正文的附加内容的编码，因而要获得Content- Type报头域中所引用的媒体类型，必须采用相应的解码机制。Content-Encoding这样用于记录文档的压缩方法，eg：Content- Encoding：gzip  
Content-Language  
Content-Language实体报头域描述了资源所用的自然语言。没有设置该域则认为实体内容将提供给所有的语言阅读  
者。eg：Content-Language:da  
Content-Length  
Content-Length实体报头域用于指明实体正文的长度，以字节方式存储的十进制数字来表示。  
Content-Type  
Content-Type实体报头域用语指明发送给接收者的实体正文的媒体类型。eg：  
Content-Type:text/html;charset=ISO-8859-1  
Content-Type:text/html;charset=GB2312  
Last-Modified  
Last-Modified实体报头域用于指示资源的最后修改日期和时间。  
Expires  
Expires实体报头域给出响应过期的日期和时间。为了让代理服务器或浏览器在一段时间以后更新缓存中(再次访问曾访问过的页面时，直接从缓存中加载， 缩短响应时间和降低服务器负载)的页面，我们可以使用Expires实体报头域指定页面过期的时间。eg：Expires：Thu，15 Sep 2006 16:23:12 GMT  
HTTP1.1的客户端和缓存必须将其他非法的日期格式（包括0）看作已经过期。eg：为了让浏览器不要缓存页面，我们也可以利用Expires实体报头域，设置为0，jsp中程序如下：response.setDateHeader("Expires","0");

### Https

我们都知道HTTPS能够加密信息，以免敏感信息被第三方获取，所以很多银行网站或电子邮箱等等安全级别较高的服务都会采用HTTPS协议。



客户端在使用HTTPS方式与Web服务器通信时有以下几个步骤，如图所示。

　　（1）客户使用https的URL访问Web服务器，要求与Web服务器建立SSL连接。

　　（2）Web服务器收到客户端请求后，会将网站的证书信息（证书中包含公钥）传送一份给客户端。

　　（3）客户端的浏览器与Web服务器开始协商SSL连接的安全等级，也就是信息加密的等级。

　　（4）客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站。

　　（5）Web服务器利用自己的私钥解密出会话密钥。

　　（6）Web服务器利用会话密钥加密与客户端之间的通信。

#### 优点

尽管HTTPS并非绝对安全，掌握根证书的机构、掌握加密算法的组织同样可以进行中间人形式的攻击，但HTTPS仍是现行架构下最安全的解决方案，主要有以下几个好处：

　　（1）使用HTTPS协议可认证用户和服务器，确保数据发送到正确的客户机和服务器；

　　（2）HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全，可防止数据在传输过程中不被窃取、改变，确保数据的完整性。

　　（3）HTTPS是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本。

　　（4）谷歌曾在2014年8月份调整搜索引擎算法，并称“比起同等HTTP网站，采用HTTPS加密的网站在搜索结果中的排名将会更高”。

#### 缺点

虽然说HTTPS有很大的优势，但其相对来说，还是存在不足之处的：

　　（1）HTTPS协议握手阶段比较费时，会使页面的加载时间延长近50%，增加10%到20%的耗电；

　　（2）HTTPS连接缓存不如HTTP高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；

　　（3）SSL证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用。

　   （4）SSL证书通常需要绑定IP，不能在同一IP上绑定多个域名，IPv4资源不可能支撑这个消耗。

　　（5）HTTPS协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL证书的信用链体系并不安全，特别是在某些国家可以控制CA根证书的情况下，中间人攻击一样可行。

#### SSL

安全套接字（Secure Socket Layer，SSL）协议是Web浏览器与Web服务器之间安全交换信息的协议，提供两个基本的安全服务：鉴别与保密。

　　SSL是Netscape于1994年开发的，后来成为了世界上最著名的web安全机制，所有主要的浏览器都支持SSL协议。

　　目前有三个版本：2、3、3.1，最常用的是第3版，是1995年发布的。

　　SSL协议的三个特性

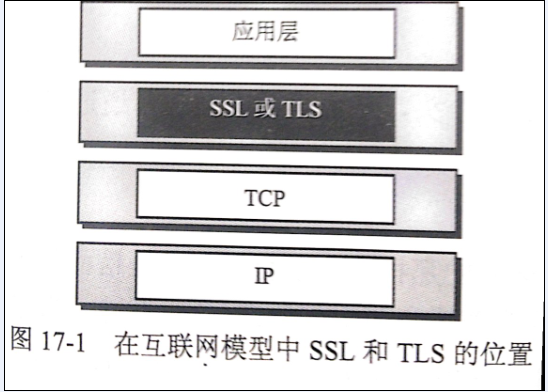
　　① 保密：在握手协议中定义了会话密钥后，所有的消息都被加密。

　　② 鉴别：可选的客户端认证，和强制的服务器端认证。

③ 完整性：传送的消息包括消息完整性检查（使用MAC）。

**SSL的位置**

　　SSL介于应用层和TCP层之间。应用层数据不再直接传递给传输层，而是传递给SSL层，SSL层对从应用层收到的数据进行加密，并增加自己的SSL头。



https://www.cnblogs.com/jeriffe/articles/2804190.html

### Http与HTTPS的区别

超文本传输协议HTTP协议被用于在Web浏览器和网站服务器之间传递信息，HTTP协议以明文方式发送内容，不提供任何方式的数据加密，如果攻击者截取了Web浏览器和网站服务器之间的传输报文，就可以直接读懂其中的信息，因此，HTTP协议不适合传输一些敏感信息，比如：信用卡号、密码等支付信息。

　　为了解决HTTP协议的这一缺陷，需要使用另一种协议：安全套接字层超文本传输协议HTTPS，为了数据传输的安全，HTTPS在HTTP的基础上加入了SSL协议，SSL依靠证书来验证服务器的身份，并为浏览器和服务器之间的通信加密。

HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。简单来说，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。

　　HTTPS和HTTP的区别主要如下：

　　1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

　　2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

　　3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

　　4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

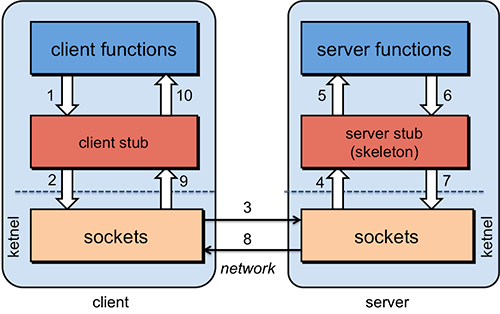
## RPC协议

RPC是指远程过程调用，也就是说两台服务器，一个应用部署在其中一台服务器上，想要调用另外一台服务器上应用提供的函数（方法），由于不在一个内存空间，不能直接调用，需要通过网络来表达调用的语义和传达调用的数据。

　　RPC 采用客户机/服务器模式。请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务端，进程保持睡眠状态直到调用信息的到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用过程接收答复信息，获得进程结果，然后调用执行继续进行。

　　实现两台服务器之间的交互，必须得通过通信协议，RPC的通讯协议实际上是使用TCP/IP协议（可以是按需连接，调用结束后就断掉，也可以是长连接，多个远程过程调用共享同一个连接），只不过RPC将其进行了封装处理，只需要调用就可以进行连接，不必关心底层，这一点与如今的框架的目的相同（将工作的中心放在了业务流程上）。那么如何找到目标服务器！仅仅通过IP是不行的，因为你无法确定目标服务器是否也同时在为其它端（泛指客户端，web端，也可以是服务端）提供其它服务，所以，需要一个标示，也就是端口，其它端就能够准确无误的找到自己所需要的进程，并通过进程号，找到目标服务，调用其提供的数据。

　　那么RPC的安全性如何？TCP/IP通信协议是基于二进制的，在内存中的参数的值要序列化成二进制的形式，也就是序列化（Serialize）或编组（marshal），通过寻址（也就是IP+POST)和传输将序列化的二进制发送,而接收方收到请求后，则需要对参数进行反序列化（序列化的逆操作），恢复为内存中的表达方式，然后找到对应的方法进行本地调用，然后得到返回值，返回值也需要经过序列化的方式发送，调用方接收到以后，再反序列化，恢复为内存中的表达方式……



　　　　　　　　　　　　　　　　图片来自网络

　　使用RPC的优点是，不需要了解底层网络技术的协议，为通信程序之间携带信息数据。在网络通信模型中，RPC 跨越了传输层和应用层。RPC 使得开发包括网络分布式多程序在内的应用程序更加容易。

## 短连接与长连接

### Http协议与TCP/IP协议的关系

HTTP的长连接和短连接本质上是TCP长连接和短连接。HTTP属于应用层协议，在传输层使用TCP协议，在网络层使用IP协议。 IP协议主要解决网络路由和寻址问题，TCP协议主要解决如何在IP层之上可靠地传递数据包，使得网络上接收端收到发送端所发出的所有包，并且顺序与发送顺序一致。TCP协议是可靠的、面向连接的。

### 如何理解http协议的无状态

HTTP协议是无状态的，指的是协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。也就是说，打开一个服务器上的网页和上一次打开这个服务器上的网页之间没有任何联系。HTTP是一个无状态的面向连接的协议，无状态不代表HTTP不能保持TCP连接，更不能代表HTTP使用的是UDP协议（无连接）。

### 什么是长连接、短连接

在HTTP/1.0中默认使用短连接。也就是说，客户端和服务器每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个HTML或其他类型的Web页中包含有其他的Web资源（如JavaScript文件、图像文件、CSS文件等），每遇到这样一个Web资源，浏览器就会重新建立一个HTTP会话。

而从HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头加入这行代码：

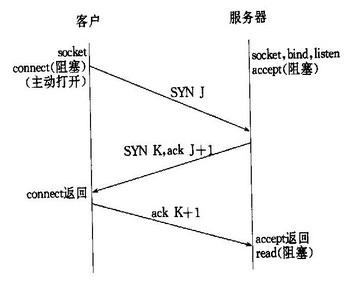
Connection:keep-alive

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接需要客户端和服务端都支持长连接。

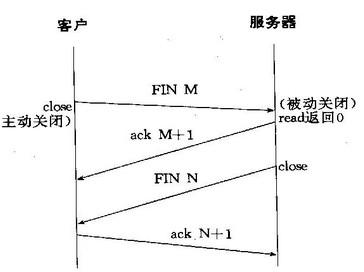
HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接。

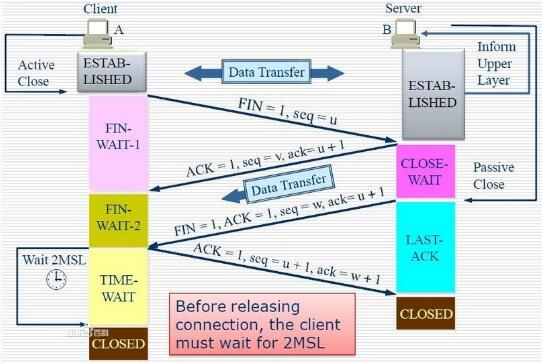
#### TCP连接

当网络通信时采用TCP协议时，在真正的读写操作之前，客户端与服务器端之间必须建立一个连接，当读写操作完成后，双方不再需要这个连接时可以释放这个连接。连接的建立依靠“三次握手”，而释放则需要“四次握手”，所以每个连接的建立都是需要资源消耗和时间消耗的。

经典的三次握手建立连接示意图：  




经典的四次握手关闭连接示意图：  




#### TCP短连接

模拟一下TCP短连接的情况：client向server发起连接请求，server接到请求，然后双方建立连接。client向server发送消息，server回应client，然后一次请求就完成了。这时候双方任意都可以发起close操作，不过一般都是client先发起close操作。上述可知，短连接一般只会在 client/server间传递一次请求操作。

短连接的优点是：管理起来比较简单，存在的连接都是有用的连接，不需要额外的控制手段。

#### TCP长连接

我们再模拟一下长连接的情况：client向server发起连接，server接受client连接，双方建立连接，client与server完成一次请求后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

TCP的保活功能主要为服务器应用提供。如果客户端已经消失而连接未断开，则会使得服务器上保留一个半开放的连接，而服务器又在等待来自客户端的数据，此时服务器将永远等待客户端的数据。保活功能就是试图在服务端器端检测到这种半开放的连接。

如果一个给定的连接在两小时内没有任何动作，服务器就向客户发送一个探测报文段，根据客户端主机响应探测4个客户端状态：

* 客户主机依然正常运行，且服务器可达。此时客户的TCP响应正常，服务器将保活定时器复位。
* 客户主机已经崩溃，并且关闭或者正在重新启动。上述情况下客户端都不能响应TCP。服务端将无法收到客户端对探测的响应。服务器总共发送10个这样的探测，每个间隔75秒。若服务器没有收到任何一个响应，它就认为客户端已经关闭并终止连接。
* 客户端崩溃并已经重新启动。服务器将收到一个对其保活探测的响应，这个响应是一个复位，使得服务器终止这个连接。
* 客户机正常运行，但是服务器不可达。这种情况与第二种状态类似。

### 短连接和长连接的缺点和优点

由上可以看出，长连接可以省去较多的TCP建立和关闭的操作，减少浪费，节约时间。对于频繁请求资源的客户端适合使用长连接。在长连接的应用场景下，client端一般不会主动关闭连接，当client与server之间的连接一直不关闭，随着客户端连接越来越多，server会保持过多连接。这时候server端需要采取一些策略，如关闭一些长时间没有请求发生的连接，这样可以避免一些恶意连接导致server端服务受损；如果条件允许则可以限制每个客户端的最大长连接数，这样可以完全避免恶意的客户端拖垮整体后端服务。

短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。但如果客户请求频繁，将在TCP的建立和关闭操作上浪费较多时间和带宽。

长连接和短连接的产生在于client和server采取的关闭策略。不同的应用场景适合采用不同的策略。

由上可以看出，**长连接**可以**省去较多的TCP建立和关闭的操作，减少浪费，节约时间**。对于频繁请求资源的客户来说，较适用长连接。不过这里**存在一个问题**，**存活功能的探测周期太长**，还有就是它只是探测TCP连接的存活，属于比较斯文的做法，遇到恶意的连接时，保活功能就不够使了。在长连接的应用场景下，client端一般不会主动关闭它们之间的连接，**Client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server早晚有扛不住的时候**，这时候server端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可 以避免一些恶意连接导致server端服务受损；如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。

**短连接**对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。但如果客户**请求频繁**，将在**TCP的建立和关闭操作上浪费时间和带宽**。

长连接和短连接的产生在于client和server采取的关闭策略，具体的应用场景采用具体的策略，没有十全十美的选择，只有合适的选择。

### 什么时候用短连接和长连接

**长连接**多用于操作频繁，点对点的通讯，而且连接数不能太多情况，。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，次处理时直接发送数据包就OK了，不用建立TCP连接。例如：数据库的连接用长连接， 如果用短连接频繁的通信会造成socket错误，而且频繁的socket 创建也是对资源的浪费。

　　而像WEB网站的http服务一般都用**短链接**，因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好。

# 网络安全知识

## 摘要算法

常用的有MD5、SHA1，摘要算法是一个不可逆过程，就是无论多大数据，经过算法运算后都是生成固定长度的数据,一般结果使用16进制进行显示。不需要密钥。

MD5和SHA1的区别：MD5结果是128位摘要，SHa1是160位摘要。那么MD5的速度更快，而SHA1的强度更高。

1、MD5：与MD5同一个家族还有MD2、MD4，都是产生128位摘要。主要用途有：验证消息完整性，安全访问认证，数据签名。

消息完整性：由于每一份数据生成的MD5值不一样，因此发送数据时可以将数据和其MD5值一起发送，然后就可以用MD5验证数据是否丢失、修改。

安全访问认证：这是使用了算法的不可逆性质，（就是无法从MD5值中恢复原数据）对账号登陆的密码进行MD5运算然后保存，这样可以保证除了用户之外，即使数据库管理人员都无法得知用户的密码。

数字签名：这是结合非对称加密算法和CA证书的一种使用场景。

加密采用Apache提供commons-codec的jar包，改jar包在springboot中dependencyManagement管理了

|  |
| --- |
| /\*\*  \* 用“jdk的实现”生成MD5摘要，在rt.jar中java.security.MessageDigest  \* 使用Apache提供commons-codec.jar的Hex做编码转换  \* MD5生成128位摘要，用32位的十六进制表示  \*  \* **@param** message  \* **@return**  \* **@throws** NoSuchAlgorithmException  \* **@throws** UnsupportedEncodingException  \*/  **public** **static** String md5Sum(String message) **throws** NoSuchAlgorithmException, UnsupportedEncodingException{  MessageDigest digest = MessageDigest.*getInstance*("MD5");  **byte**[] bytes = digest.digest(message.getBytes("UTF-8"));  String md5 = Hex.*encodeHexString*(bytes);  **return** md5;  } |

|  |
| --- |
| /\*\*  \* 使用Apache提供commons-codec.jar做md5摘要  \*  \* **@param** message  \* **@return**  \*/  **public** **static** String md5Sum2(String message){  **return** org.apache.commons.codec.digest.DigestUtils.*md5Hex*(message);  } |

一般破解方法：字典法，就是将常用密码生成MD5值字典，然后反向查找达到破解目的，因此建议使用强密码。

2、SHA1：SHA1生产160位摘要，以16进制显示40位。与SHA1同一个家族还有SHA224、SHA256、SHA384、SHA512，产生的位数不一样。

|  |
| --- |
| /\*\*  \* 使用jdk提供生成SHA1摘要。  \* 使用Apache提供commons-codec.jar的Hex做编码转换  \* SHA1生成160位摘要，SHA256生成256位摘要，SHA384生成384位摘要，SHA512生成512位摘要  \*  \* **@param** message  \* **@return**  \* **@throws** NoSuchAlgorithmException  \*/  **public** **static** String sha1Sum(String message) **throws** NoSuchAlgorithmException{  MessageDigest digest = MessageDigest.*getInstance*("SHA-1");  **byte**[] bytes = digest.digest(message.getBytes());  String sha1 = Hex.*encodeHexString*(bytes);  **return** sha1;  } |

|  |
| --- |
| /\*\*  \* 使用Apache提供commons-codec.jar做sha1摘要  \*  \* **@param** message  \* **@return**  \*/  **public** **static** String sha1Sum2(String message){  **return** org.apache.commons.codec.digest.DigestUtils.*sha1Hex*(message);  } |

## 签名与验签

SHA1withRSA做签名，先进行SHA1签名，再对签名结果在做RSA加密。

类似还有MD5withRSA.

|  |
| --- |
| /\*\*  \* SHA1withRSA做签名，先进行SHA1签名，再对签名结果在做RSA加密。  \* 类似还有MD5withRSA.  \*  \*/  **public** **class** SHA1withRSAUtils {  **private** **static** **final** String ***KEY\_ALGORITHM*** = "SHA1withRSA";  //private static final String KEY\_ALGORITHM = "MD5withRSA";    **private** **static** **final** String ***ENCODING*** = "UTF-8";    /\*\*  \* 签名  \*  \* **@param** data  \* **@param** rsaPrivateKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** **byte**[] sign(String data, RSAPrivateKey rsaPrivateKey) **throws** Exception{  //获取Signature对象  Signature sign = Signature.*getInstance*(***KEY\_ALGORITHM***);  //初始化sign对象  sign.initSign(rsaPrivateKey);  //签名  sign.update(data.getBytes(***ENCODING***));  **byte**[] bytes = sign.sign();    **return** bytes;  }    /\*\*  \* 验签  \*  \* **@param** bytes  \* **@param** data  \* **@param** rsaPublicKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** **boolean** verify(**byte**[] bytes, String data, RSAPublicKey rsaPublicKey) **throws** Exception{  //获取Signature对象  Signature sign = Signature.*getInstance*(***KEY\_ALGORITHM***);  //初始化sign对象  sign.initVerify(rsaPublicKey);  //验签  sign.update(data.getBytes(***ENCODING***));  **return** sign.verify(bytes);  }    **public** **static** **void** main(String[] args) **throws** Exception {  //密钥对生产：http://web.chacuo.net/netrsakeypair  String priKey = "";  String pubKey = "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDWlJp3G5MOxfBYh7X+2RjpdQkT5D6OFKapCsxwVYmLOtYgE79cbsoS6HaQMURzLcJaosjfNY9OGE7kJZKf3Lezv7dVO/0hE9BCEJIeIgrwo6yW+KymHN0caV7NSwJRw1o73LrLGpBYyr44sWpFA6VuZDcEhrAsIndDLiotkniF7wIDAQAB";    String message = "rsa";    RSAPrivateKey rsaPrivateKey = RSAUtils.*getRSAPrivateKey*(priKey);  RSAPublicKey rsaPublicKey = RSAUtils.*getRSAPublicKey*(pubKey);    **byte**[] bytes = *sign*(message, rsaPrivateKey);  **boolean** verify = *verify*(bytes, message, rsaPublicKey);  System.***out***.println(verify);  }  } |

## 对称加密

对称加密是加密与解密使用相同的密钥。常见算法有AES、DES

对称加密与非对称加密区别：对称加密算法只是为了区分非对称加密算法。其中鲜明的特点是对称加密是加密解密使用相同的密钥，而非对称加密加密和解密时使用的密钥不一样。对于大部分情况我们都使用对称加密，而对称加密的密钥交换时使用非对称加密，这有效保护密钥的安全。非对称加密加密和解密密钥不同，那么它的安全性是无疑最高的，但是它加密解密的速度很慢，不适合对大数据加密。而对称加密加密速度快，因此混合使用最好。

**对称加密是最快速、最简单的一种加密方式，加密（encryption）与解密（decryption）用的是同样的密钥（secret key）。对称加密有很多种算法，由于它效率很高，所以被广泛使用在很多加密协议的核心当中。**

**对称加密通常使用的是相对较小的密钥，一般小于256 bit。因为密钥越大，加密越强，但加密与解密的过程越慢。如果你只用1 bit来做这个密钥，那黑客们可以先试着用0来解密，不行的话就再用1解；但如果你的密钥有1 MB大，黑客们可能永远也无法破解，但加密和解密的过程要花费很长的时间。密钥的大小既要照顾到安全性，也要照顾到效率，是一个trade-off**。

DES：比较老的算法，一共有三个参数入口（原文，密钥，加密模式）。而3DES只是DES的一种模式，是以DES为基础更安全的变形，对数据进行了三次加密，也是被指定为AES的过渡算法。

DES：已经被破解，已经不用了。

AES:高级加密标准，新一代标准，加密速度更快，安全性更高（不用说优先选择）

参数：”AES/ECB/PKCS5Padding”在加密和解密时必须相同，可以直接写”AES”,这样就是使用默认模式（C#和java默认的模式不一样，C#中默认的是这种,java的默认待研究）。分别的意思为：AES是加密算法，ECB是工作模式，PKCS5Padding是填充方式。

AES是分组加密算法，也称块加密。每一组16字节。这样明文就会分成多块。当有一块不足16字节时就会进行填充。

一共有四种工作模式：

ECB 电子密码本模式：相同的明文块产生相同的密文块，容易并行运算，但也可能对明文进行攻击。

CBC 加密分组链接模式：一块明文加密后和上一块密文进行链接，不利于并行，但安全性比ECB好，是SSL,IPSec的标准。

CFB 加密反馈模式：将上一次密文与密钥运算，再加密。隐藏明文模式，不利于并行，误差传递。

OFB 输出反馈模式：将上一次处理过的密钥与密钥运算，再加密。隐藏明文模式，不利于并行，有可能明文攻击，误差传递。

PKCS5Padding的填充方式是差多少字节就填数字多少；刚好每一不足16字节时，那么就会加一组填充为16.还有其他填充模式【Nopadding,ISO10126Padding】（不影响算法，加密解密时一致就行）。

|  |
| --- |
| /\*\*  \* 对称加密，已经被破解，不再使用。  \* 3DES是DES升级版  \*  \*/  **public** **class** DESUtils {    **private** **static** **final** String ***DEFAULT\_CIPHER\_ALGORITHM*** = "DES/ECB/PKCS5Padding";  **private** **static** **final** String ***KEY\_ALGORITHM*** = "DES";    **private** **static** **final** String ***ENCODING*** = "UTF-8";    **public** **static** **byte**[] aesEncrypt(**final** String message, **final** String key) **throws** Exception{  //获取cipher对象  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  //初始化cipher对象  cipher.init(Cipher.***ENCRYPT\_MODE***, *getSecretKey*(key), **new** SecureRandom());  //执行操作  **byte**[] bytes = cipher.doFinal(message.getBytes(***ENCODING***));  **return** bytes;  }    **public** **static** String aesDecrypt(**final** **byte**[] bytes, **final** String key) **throws** Exception{  //获取cipher对象  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  //初始化cipher对象  cipher.init(Cipher.***DECRYPT\_MODE***, *getSecretKey*(key), **new** SecureRandom());  //执行操作  **byte**[] results = cipher.doFinal(bytes);  **return** **new** String(results, ***ENCODING***);  }    **private** **static** SecretKey getSecretKey(**final** String key) **throws** Exception{  // 从原始密钥数据创建DESKeySpec对象  DESKeySpec dks = **new** DESKeySpec(key.getBytes(***ENCODING***));  // 创建一个密钥工厂，然后用它把DESKeySpec转换成SecretKey对象  SecretKeyFactory factory = SecretKeyFactory.*getInstance*(***KEY\_ALGORITHM***);  SecretKey secretKey = factory.generateSecret(dks);  **return** secretKey;  }    **public** **static** **void** main(String[] args) **throws** Exception {  String message = "des";  String key = "12345678";//密码一定要是8的倍数  **byte**[] bytes = *aesEncrypt*(message, key);  System.***out***.println(**new** String(bytes, ***ENCODING***));    String result = *aesDecrypt*(bytes, key);  System.***out***.println(result);  }  } |

|  |
| --- |
| /\*\*  \* 最常用的对称加密算法  \*  \*/  **public** **class** AESUtils {    /\*\*  \* 加密和解密必须用相同  \*  \* AES：加密算法  \* ECB：工作模式  \* PKCS5Padding：填充方式  \*/  **private** **static** **final** String ***DEFAULT\_CIPHER\_ALGORITHM*** = "AES/ECB/PKCS5Padding";    **private** **static** **final** String ***KEY\_ALGORITHM*** = "AES";    **private** **static** **final** String ***ENCODING*** = "UTF-8";  /\*\*  \* **@param** message  \* **@param** key  \* **@return**  \* **@throws** Exception  \*/  **public** **static** **byte**[] aesEncrypt(**final** String message, **final** String key) **throws** Exception{  //获取Cipher对象  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  //使用密钥初始化Cipher，设置opmode=1(解密)  cipher.init(Cipher.***ENCRYPT\_MODE***, *getSecretKey*(key));  //执行操作  **byte**[] bytes = cipher.doFinal(message.getBytes("UTF-8"));    **return** bytes;  }    /\*\*  \* **@param** bytes  \* **@param** key  \* **@return**  \* **@throws** Exception  \*/  **public** **static** String aesDecrypt(**final** **byte**[] bytes, **final** String key) **throws** Exception{  //获取cipher对象  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  //使用密钥初始化cipher，设置opmode=2(解密)  cipher.init(Cipher.***DECRYPT\_MODE***, *getSecretKey*(key));  //执行操作  **byte**[] results = cipher.doFinal(bytes);  //byte数组转字符串  String message = **new** String(results, ***ENCODING***);  **return** message;  }    **private** **static** SecretKeySpec getSecretKey(**final** String key) **throws** Exception {  //返回生成指定算法密钥生成器的 KeyGenerator 对象  KeyGenerator keyGenerator = KeyGenerator.*getInstance*(***KEY\_ALGORITHM***);  //AES 要求密钥长度为 128  keyGenerator.init(128, **new** SecureRandom(key.getBytes(***ENCODING***)));  //生成一个密钥  SecretKey secretKey = keyGenerator.generateKey();    **return** **new** SecretKeySpec(secretKey.getEncoded(), ***KEY\_ALGORITHM***);  }    **public** **static** **void** main(String[] args) **throws** Exception {  String message = "aes";  String key = "1";  **byte**[] bytes = *aesEncrypt*(message, key);  System.***out***.println(**new** String(bytes, "UTF-8"));    String result = *aesDecrypt*(bytes, key);  System.***out***.println(result);  }  } |

## 非对称加密

加密跟解密使用不同的密钥。常见算法：RSA

1. RSA加密：公钥加密，私钥解密。私钥加密、公钥解密
2. SHA1WithRSA签名：公钥签名、私钥验签

|  |
| --- |
| /\*\*  \* 非对称加密：RSA  \*  \*/  **public** **class** RSAUtils {    **private** **static** **final** String ***DEFAULT\_CIPHER\_ALGORITHM*** = "RSA/ECB/PKCS1Padding";    **private** **static** **final** String ***KEY\_ALGORITHM*** = "RSA";    **private** **static** **final** String ***ENCODING*** = "UTF-8";    /\*\*  \* 根据公钥字符串生产公钥  \*  \* **@param** publicKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** RSAPublicKey getRSAPublicKey(String publicKey) **throws** Exception{  KeyFactory factory = KeyFactory.*getInstance*(***KEY\_ALGORITHM***);  X509EncodedKeySpec spec = **new** X509EncodedKeySpec(Base64.*decode*(publicKey));  RSAPublicKey rsaPublicKey = (RSAPublicKey) factory.generatePublic(spec);  **return** rsaPublicKey;  }    /\*\*  \* 根据私钥字符串生产私钥  \*  \* **@param** privateKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** RSAPrivateKey getRSAPrivateKey(String privateKey) **throws** Exception{  KeyFactory factory = KeyFactory.*getInstance*(***KEY\_ALGORITHM***);  PKCS8EncodedKeySpec keySpec = **new** PKCS8EncodedKeySpec(Base64.*decode*(privateKey));  RSAPrivateKey rsaPrivateKey = (RSAPrivateKey) factory.generatePrivate(keySpec);  **return** rsaPrivateKey;  }    /\*\*  \* 公钥加密  \*  \* **@param** data  \* **@param** rsaPublicKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** **byte**[] publicEncrypt(String data, RSAPublicKey rsaPublicKey) **throws** Exception{  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  cipher.init(Cipher.***ENCRYPT\_MODE***, rsaPublicKey);    **int** maxBlock = rsaPublicKey.getModulus().bitLength()/8 -11;  ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();  **int** offset = 0;  **byte**[] buff;  **int** i = 0;  **byte**[] datas = data.getBytes(***ENCODING***);  **while**(datas.length > offset){  **if**(datas.length-offset > maxBlock){  buff = cipher.doFinal(datas, offset, maxBlock);  }**else**{  buff = cipher.doFinal(datas, offset, datas.length-offset);  }  byteArrayOutputStream.write(buff, 0, buff.length);  i++;  offset = i \* maxBlock;  }  **byte**[] resultDatas = byteArrayOutputStream.toByteArray();  **return** resultDatas;  }    /\*\*  \* 私钥解密  \*  \* **@param** datas  \* **@param** rsaPrivateKey  \* **@return**  \*/  **public** **static** **byte**[] privateDecrypt(**byte**[] datas, RSAPrivateKey rsaPrivateKey) **throws** Exception{  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  cipher.init(Cipher.***DECRYPT\_MODE***, rsaPrivateKey);    **int** maxBlock = rsaPrivateKey.getModulus().bitLength()/8;  ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();  **int** offset = 0;  **byte**[] buff;  **int** i = 0;  **while**(datas.length > offset){  **if**(datas.length-offset > maxBlock){  buff = cipher.doFinal(datas, offset, maxBlock);  }**else**{  buff = cipher.doFinal(datas, offset, datas.length-offset);  }  byteArrayOutputStream.write(buff, 0, buff.length);  i++;  offset = i \* maxBlock;  }  **byte**[] resultDatas = byteArrayOutputStream.toByteArray();  **return** resultDatas;  }    //------------------------------------------------------------------------------    /\*\*  \* 私钥加密  \*  \* **@param** data  \* **@param** rsaPrivateKey  \* **@return**  \*/  **public** **static** **byte**[] privateEncrypt(String data, RSAPrivateKey rsaPrivateKey) **throws** Exception{  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  cipher.init(Cipher.***ENCRYPT\_MODE***, rsaPrivateKey);    **int** maxBlock = rsaPrivateKey.getModulus().bitLength()/8 -11;  ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();  **int** offset = 0;  **byte**[] buff;  **int** i = 0;  **byte**[] datas = data.getBytes(***ENCODING***);  **while**(datas.length > offset){  **if**(datas.length-offset > maxBlock){  buff = cipher.doFinal(datas, offset, maxBlock);  }**else**{  buff = cipher.doFinal(datas, offset, datas.length-offset);  }  byteArrayOutputStream.write(buff, 0, buff.length);  i++;  offset = i \* maxBlock;  }  **byte**[] resultDatas = byteArrayOutputStream.toByteArray();  **return** resultDatas;  }    /\*\*  \* 公钥解密  \*  \* **@param** datas  \* **@param** rsaPublicKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** **byte**[] publicDecrypt(**byte**[] datas, RSAPublicKey rsaPublicKey) **throws** Exception{  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  cipher.init(Cipher.***DECRYPT\_MODE***, rsaPublicKey);    **int** maxBlock = rsaPublicKey.getModulus().bitLength()/8;  ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();  **int** offset = 0;  **byte**[] buff;  **int** i = 0;  **while**(datas.length > offset){  **if**(datas.length-offset > maxBlock){  buff = cipher.doFinal(datas, offset, maxBlock);  }**else**{  buff = cipher.doFinal(datas, offset, datas.length-offset);  }  byteArrayOutputStream.write(buff, 0, buff.length);  i++;  offset = i \* maxBlock;  }  **byte**[] resultDatas = byteArrayOutputStream.toByteArray();  **return** resultDatas;  }    **public** **static** **void** main(String[] args) **throws** Exception {  //密钥对生产：http://web.chacuo.net/netrsakeypair  String priKey = "";  String pubKey = "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDWlJp3G5MOxfBYh7X+2RjpdQkT5D6OFKapCsxwVYmLOtYgE79cbsoS6HaQMURzLcJaosjfNY9OGE7kJZKf3Lezv7dVO/0hE9BCEJIeIgrwo6yW+KymHN0caV7NSwJRw1o73LrLGpBYyr44sWpFA6VuZDcEhrAsIndDLiotkniF7wIDAQAB";    String message = "rsa";    RSAPrivateKey rsaPrivateKey = *getRSAPrivateKey*(priKey);  RSAPublicKey rsaPublicKey = *getRSAPublicKey*(pubKey);    **byte**[] bytes1 = *publicEncrypt*(message, rsaPublicKey);  **byte**[] datas1 = *privateDecrypt*(bytes1, rsaPrivateKey);  System.***out***.println(**new** String(datas1, ***ENCODING***));    **byte**[] bytes2 = *privateEncrypt*(message, rsaPrivateKey);  **byte**[] datas2 = *publicDecrypt*(bytes2, rsaPublicKey);  System.***out***.println(**new** String(datas2, ***ENCODING***));  } |

## 编码算法

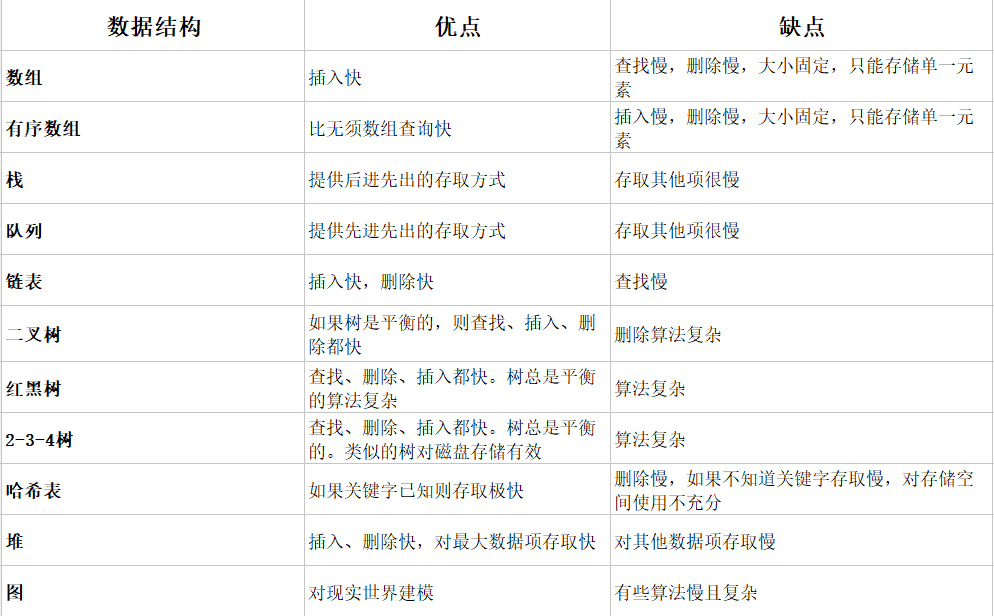
常见的编码有Base64,HEX和对URL的编码。这都是为了实际需要才进行的编码。HEX是编码成16进制字符，MD5一般就是以HEX进行编码，这不说了。

|  |
| --- |
| /\*\*  \* base64编码解码。  \* 在jdk1.8之前，用sun.misc.Base64Encoder和sun.misc.Base64Decoder，能力较差。  \* 在jdk1.8之前，通常用Apache的commons-codec.jar中Base64来做编码。  \*  \* 在jdk1.8，jdk提供了java.util.Base64.Decoder和java.util.Base64.Encoder做编码  \* 结论：如果是jdk1.8之前，用Apache来做编码；  \* 如果是jdk1.8，用jdk提供包来做编码  \*  \*/  **public** **class** Base64Utils {    **public** **static** String encode(**byte**[] bytes){  Encoder encoder = Base64.*getEncoder*();  **return** encoder.encodeToString(bytes);  }    **public** **static** **byte**[] decode(String data){  Decoder decoder = Base64.*getDecoder*();  **return** decoder.decode(data);  }    **public** **static** **void** main(String[] args) **throws** UnsupportedEncodingException {  String data = "base64";    String base = *encode*(data.getBytes("UTF-8"));  System.***out***.println(base);    System.***out***.println(**new** String(*decode*(base), "UTF-8"));  }  } |

|  |
| --- |
| /\*\*  \* URL编码解码  \*  \*/  **public** **class** URLUtils {    **private** **static** **final** String ***ENCODING*** = "UTF-8";  /\*\*  \* 编码  \*  \* **@param** src  \* **@return**  \* **@throws** UnsupportedEncodingException  \*/  **public** **static** String encoder(String src) **throws** UnsupportedEncodingException{  String des = URLEncoder.*encode*(src, ***ENCODING***);  **return** des;  }    /\*\*  \* 解码  \*  \* **@param** src  \* **@return**  \* **@throws** UnsupportedEncodingException  \*/  **public** **static** String decoder(String src) **throws** UnsupportedEncodingException{  String des = URLDecoder.*decode*(src, ***ENCODING***);  **return** des;  }    **public** **static** **void** main(String[] args) **throws** UnsupportedEncodingException {  String url = "http://baidu.com/对的";  String encoder = *encoder*(url);  System.***out***.println(encoder);    String decoder = *decoder*(encoder);  System.***out***.println(decoder);  }  } |

# 算法与数据结构

## 数据结构



### 表（列表）

#### 数组

##### Java数组

在Java中，数组是用来存放同一种数据类型的集合，注意只能存放同一种数据类型(Object类型数组除外)。

**①、数组的声明**

**第一种方式：**

|  |  |
| --- | --- |
| 1 | 数据类型 []  数组名称 = new 数据类型[数组长度]; |

　　这里 [] 可以放在数组名称的前面，也可以放在数组名称的后面，我们推荐放在数组名称的前面，这样看上去 数据类型 [] 表示的很明显是一个数组类型，而放在数组名称后面，则不是那么直观。

**第二种方式：**

|  |  |
| --- | --- |
| 1 | 数据类型 [] 数组名称 = {数组元素1，数组元素2，......} |

　　这种方式声明数组的同时直接给定了数组的元素，数组的大小由给定的数组元素个数决定。

|  |  |
| --- | --- |
| 1  2  3  4 | //声明数组1,声明一个长度为3，只能存放int类型的数据  int [] myArray = new int[3];  //声明数组2,声明一个数组元素为 1,2,3的int类型数组  int [] myArray2 = {1,2,3}; |

**②、访问数组元素以及给数组元素赋值**

　　数组是存在下标索引的，通过下标可以获取指定位置的元素，数组小标是从0开始的，也就是说下标0对应的就是数组中第1个元素，可以很方便的对数组中的元素进行存取操作。

　　前面数组的声明第二种方式，我们在声明数组的同时，也进行了初始化赋值。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //声明数组,声明一个长度为3，只能存放int类型的数据  int [] myArray = new int[3];  //给myArray第一个元素赋值1  myArray[0] = 1;  //访问myArray的第一个元素  System.out.println(myArray[0]); |

　　上面的myArray 数组，我们只能赋值三个元素，也就是下标从0到2，如果你访问 myArray[3] ，那么会报数组下标越界异常。

**③、数组遍历**

数组有个 length 属性，是记录数组的长度的，我们可以利用length属性来遍历数组。

|  |  |
| --- | --- |
| 1  2  3  4  5 | //声明数组2,声明一个数组元素为 1,2,3的int类型数组  int [] myArray2 = {1,2,3};  for(int i = 0 ; i < myArray2.length ; i++){      System.out.println(myArray2[i]);  } |

##### 分析数组的局限性

通过上面的代码，我们发现数组是能完成一个数据结构所有的功能的，而且实现起来也不难，那数据既然能完成所有的工作，我们实际应用中为啥不用它来进行所有的数据存储呢？那肯定是有原因呢。

　　数组的局限性分析：

　　①、插入快，对于无序数组，上面我们实现的数组就是无序的，即元素没有按照从大到小或者某个特定的顺序排列，只是按照插入的顺序排列。无序数组增加一个元素很简单，只需要在数组末尾添加元素即可，但是有序数组却不一定了，它需要在指定的位置插入。

　　②、查找慢，当然如果根据下标来查找是很快的。但是通常我们都是根据元素值来查找，给定一个元素值，对于无序数组，我们需要从数组第一个元素开始遍历，直到找到那个元素。有序数组通过特定的算法查找的速度会比无需数组快，后面我们会讲各种排序算法。

　　③、删除慢，根据元素值删除，我们要先找到该元素所处的位置，然后将元素后面的值整体向前面移动一个位置。也需要比较多的时间。

　　④、数组一旦创建后，大小就固定了，不能动态扩展数组的元素个数。如果初始化你给一个很大的数组大小，那会白白浪费内存空间，如果给小了，后面数据个数增加了又添加不进去了。

　　很显然，数组虽然插入快，但是查找和删除都比较慢，而且扩展性差，所以我们一般不会用数组来存储数据，那有没有什么数据结构插入、查找、删除都很快，而且还能动态扩展存储个数大小呢，答案是有的，但是这是建立在很复杂的算法基础上，后面我们也会详细讲解。

#### 链表

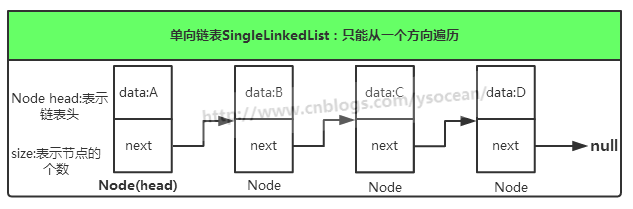
**链表**（Linked list）是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存到下一个节点的指针(Pointer)。

　　使用链表结构可以克服数组链表需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。

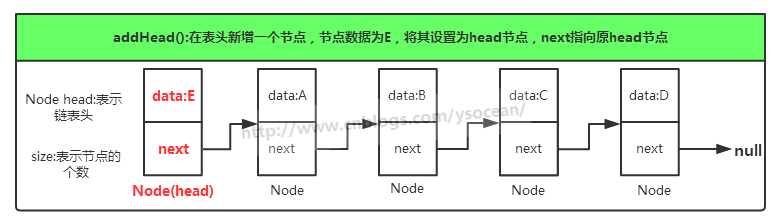
##### 单向链表

单链表是链表中结构最简单的。一个单链表的节点(Node)分为两个部分，第一个部分(data)保存或者显示关于节点的信息，另一个部分存储下一个节点的地址。最后一个节点存储地址的部分指向空值。

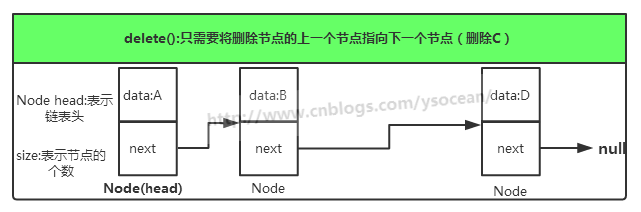
　　单向链表只可向一个方向遍历，一般查找一个节点的时候需要从第一个节点开始每次访问下一个节点，一直访问到需要的位置。而插入一个节点，对于单向链表，我们只提供在链表头插入，只需要将当前插入的节点设置为头节点，next指向原头节点即可。删除一个节点，我们将该节点的上一个节点的next指向该节点的下一个节点。



　　在表头增加节点：

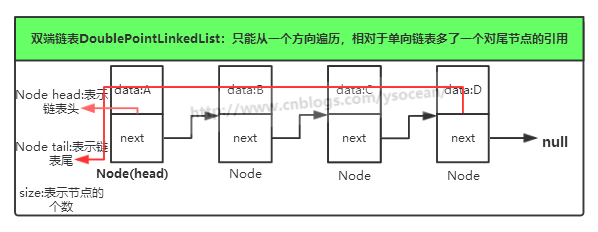


　　删除节点：



##### 双端链表

对于单项链表，我们如果想在尾部添加一个节点，那么必须从头部一直遍历到尾部，找到尾节点，然后在尾节点后面插入一个节点。这样操作很麻烦，如果我们在设计链表的时候多个对尾节点的引用，那么会简单很多。



**注意和后面将的双向链表的区别！！！**

##### 有序链表

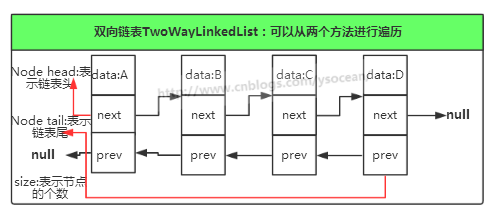
前面的链表实现插入数据都是无序的，在有些应用中需要链表中的数据有序，这称为有序链表。

　　在有序链表中，数据是按照关键值有序排列的。一般在大多数需要使用有序数组的场合也可以使用有序链表。有序链表优于有序数组的地方是插入的速度（因为元素不需要移动），另外链表可以扩展到全部有效的使用内存，而数组只能局限于一个固定的大小中。

　　在有序链表中插入和删除某一项最多需要O(N)次比较，平均需要O(N/2)次，因为必须沿着链表上一步一步走才能找到正确的插入位置，然而可以最快速度删除最值，因为只需要删除表头即可，如果一个应用需要频繁的存取最小值，且不需要快速的插入，那么有序链表是一个比较好的选择方案。比如优先级队列可以使用有序链表来实现。

##### 双向链表

我们知道单向链表只能从一个方向遍历，那么双向链表它可以从两个方向遍历。



##### 总结

上面我们讲了各种链表，每个链表都包括一个LinikedList对象和许多Node对象，LinkedList对象通常包含头和尾节点的引用，分别指向链表的第一个节点和最后一个节点。而每个节点对象通常包含数据部分data，以及对上一个节点的引用prev和下一个节点的引用next，只有下一个节点的引用称为单向链表，两个都有的称为双向链表。next值为null则说明是链表的结尾，如果想找到某个节点，我们必须从第一个节点开始遍历，不断通过next找到下一个节点，直到找到所需要的。栈和队列都是ADT，可以用数组来实现，也可以用链表实现。

### 栈

#### 栈的基本概念

**栈**（英语：stack）又称为**堆栈**或**堆叠，**栈作为一种数据结构，是一种只能在一端进行插入和删除操作的特殊线性表。它按照先进后出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个数据被第一个读出来）。栈具有记忆作用，对栈的插入与删除操作中，不需要改变栈底指针。

　　栈是允许在同一端进行插入和删除操作的特殊线性表。允许进行插入和删除操作的一端称为栈顶(top)，另一端为栈底(bottom)；栈底固定，而栈顶浮动；栈中元素个数为零时称为空栈。插入一般称为进栈（PUSH），删除则称为退栈（POP）。

　　由于堆叠数据结构只允许在一端进行操作，因而按照后进先出（LIFO, Last In First Out）的原理运作。栈也称为后进先出表。

　　这里以羽毛球筒为例，羽毛球筒就是一个栈，刚开始羽毛球筒是空的，也就是空栈，然后我们一个一个放入羽毛球，也就是一个一个push进栈，当我们需要使用羽毛球的时候，从筒里面拿，也就是pop出栈，但是第一个拿到的羽毛球是我们最后放进去的。

#### 利用栈实现字符串的逆序

我们知道栈是后进先出，我们可以将一个字符串分隔为单个的字符，然后将字符一个一个push()进栈，在一个一个pop()出栈就是逆序显示了。如下：

　　将 字符串“how are you” 反转！！！

　　ps：这里我们是用上面自定的栈来实现的，大家可以将ArrayStack替换为JDK自带的栈类Stack试试

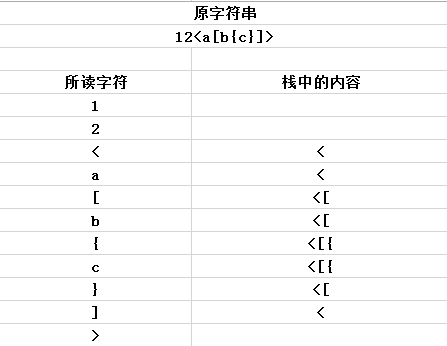
|  |
| --- |
| **package** org.xzh.springboot.datastruct;  **import** java.util.Stack;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** MainStack {    **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(MainStack.**class**);  **public** **static** String nixun(**final** String res){  StringBuffer result = **new** StringBuffer();    Stack<Character> stack = **new** Stack<Character>();  **char**[] chars = res.toCharArray();  **for**(**char** c : chars){  stack.push(c);  }    **while**(!stack.isEmpty()){  result.append(stack.pop());  }    **return** result.toString();  }    **public** **static** **void** main(String[] args) {  String result = MainStack.*nixun*("are you ok");  ***logger***.info(result);  }  } |

#### 利用栈判断分隔符是否匹配

写过xml标签或者html标签的，我们都知道<必须和最近的>进行匹配，[ 也必须和最近的 ] 进行匹配。

　　比如：<abc[123]abc>这是符号相匹配的，如果是 <abc[123>abc] 那就是不匹配的。

　　对于 12<a[b{c}]>，我们分析在栈中的数据：遇到匹配正确的就消除



　　最后栈中的内容为空则匹配成功，否则匹配失败！！！

#### 总结

根据栈后进先出的特性，我们实现了单词逆序以及分隔符匹配。所以其实栈是一个概念上的工具，具体能实现什么功能可以由我们去想象。栈通过提供限制性的访问方法push()和pop()，使得程序不容易出错。

　　对于栈的实现，我们稍微分析就知道，数据入栈和出栈的时间复杂度都为O(1)，也就是说栈操作所耗的时间不依赖栈中数据项的个数，因此操作时间很短。而且需要注意的是栈不需要比较和移动操作，我们不要画蛇添足。

### 队列

#### 队列的基本概念

队列（queue）是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。

　　队列的数据元素又称为队列元素。在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先出（FIFO—first in first out）线性表。

　　比如我们去电影院排队买票，第一个进入排队序列的都是第一个买到票离开队列的人，而最后进入排队序列排队的都是最后买到票的。

　　在比如在计算机操作系统中，有各种队列在安静的工作着，比如打印机在打印列队中等待打印。

　　队列分为：

　　①、单向队列（Queue）：只能在一端插入数据，另一端删除数据。

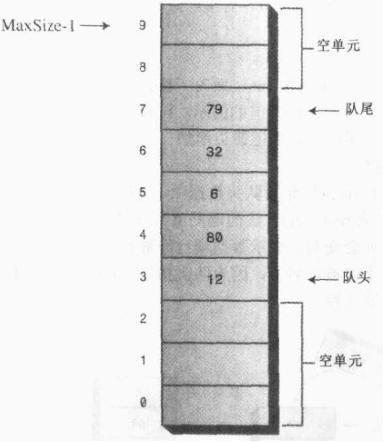
　　②、双向队列（Deque）：每一端都可以进行插入数据和删除数据操作。

　　这里我们还会介绍一种队列——优先级队列，优先级队列是比栈和队列更专用的数据结构，在优先级队列中，数据项按照关键字进行排序，关键字最小（或者最大）的数据项往往在队列的最前面，而数据项在插入的时候都会插入到合适的位置以确保队列的有序。

#### Java模拟单向队列的实现

在实现之前，我们先看下面几个问题：

　　①、与栈不同的是，队列中的数据不总是从数组的0下标开始的，移除一些队头front的数据后，队头指针会指向一个较高的下标位置，如下图：

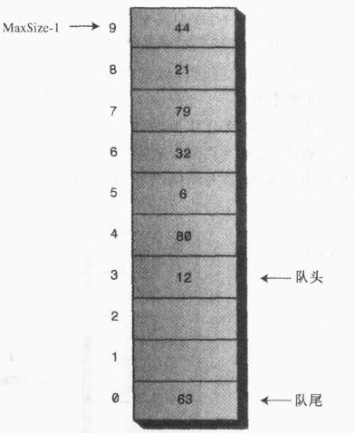


　　②、我们再设计时，队列中新增一个数据时，队尾的指针rear 会向上移动，也就是向下标大的方向。移除数据项时，队头指针 front 向上移动。那么这样设计好像和现实情况相反，比如排队买电影票，队头的买完票就离开了，然后队伍整体向前移动。在计算机中也可以在队列中删除一个数之后，队列整体向前移动，但是这样做效率很差。我们选择的做法是移动队头和队尾的指针。

　　③、如果向第②步这样移动指针，相信队尾指针很快就移动到数据的最末端了，这时候可能移除过数据，那么队头会有空着的位置，然后新来了一个数据项，由于队尾不能再向上移动了，那该怎么办呢？如下图：



　　为了避免队列不满却不能插入新的数据，我们可以让队尾指针绕回到数组开始的位置，这也称为“循环队列”。



　　弄懂原理之后，Java实现代码如下：

#### 双向队列

双端队列就是一个两端都是结尾或者开头的队列， 队列的每一端都可以进行插入数据项和移除数据项，这些方法可以叫做：

　　insertRight()、insertLeft()、removeLeft()、removeRight()

　　如果严格禁止调用insertLeft()和removeLeft()（或禁用右端操作），那么双端队列的功能就和前面讲的栈功能一样。

　　如果严格禁止调用insertLeft()和removeRight(或相反的另一对方法)，那么双端队列的功能就和单向队列一样了。

#### 优先级队列

优先级队列（priority queue）是比栈和队列更专用的数据结构，在优先级队列中，数据项按照关键字进行排序，关键字最小（或者最大）的数据项往往在队列的最前面，而数据项在插入的时候都会插入到合适的位置以确保队列的有序。

　　优先级队列 是0个或多个元素的集合，每个元素都有一个优先权，对优先级队列执行的操作有：

　　（1）查找

　　（2）插入一个新元素

　　（3）删除

　　一般情况下，查找操作用来搜索优先权最大的元素，删除操作用来删除该元素 。对于优先权相同的元素，可按先进先出次序处理或按任意优先权进行。

　　这里我们用数组实现优先级队列，这种方法插入比较慢，但是它比较简单，适用于数据量比较小并且不是特别注重插入速度的情况。

　　后面我们会讲解堆，用堆的数据结构来实现优先级队列，可以相当快的插入数据。

**数组实现优先级队列，声明为int类型的数组，关键字是数组里面的元素，在插入的时候按照从大到小的顺序排列，也就是越小的元素优先级越高。**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | package com.ys.datastructure;    public class PriorityQue {      private int maxSize;      private int[] priQueArray;      private int nItems;        public PriorityQue(int s){          maxSize = s;          priQueArray = new int[maxSize];          nItems = 0;      }        //插入数据      public void insert(int value){          int j;          if(nItems == 0){              priQueArray[nItems++] = value;          }else{              j = nItems -1;              //选择的排序方法是插入排序，按照从大到小的顺序排列，越小的越在队列的顶端              while(j >=0 && value > priQueArray[j]){                  priQueArray[j+1] = priQueArray[j];                  j--;              }              priQueArray[j+1] = value;              nItems++;          }      }        //移除数据,由于是按照大小排序的，所以移除数据我们指针向下移动      //被移除的地方由于是int类型的，不能设置为null，这里的做法是设置为 -1      public int remove(){          int k = nItems -1;          int value = priQueArray[k];          priQueArray[k] = -1;//-1表示这个位置的数据被移除了          nItems--;          return value;      }        //查看优先级最高的元素      public int peekMin(){          return priQueArray[nItems-1];      }        //判断是否为空      public boolean isEmpty(){          return (nItems == 0);      }        //判断是否满了      public boolean isFull(){          return (nItems == maxSize);      }    } |

　　insert() 方法，先检查队列中是否有数据项，如果没有，则直接插入到下标为0的单元里，否则，从数组顶部开始比较，找到比插入值小的位置进行插入，并把 nItems 加1.

　　remove 方法直接获取顶部元素。

　　优先级队列的插入操作需要 O(N)的时间，而删除操作则需要O(1) 的时间，后面会讲解如何通过 堆 来改进插入时间。

#### 总结

本篇博客我们介绍了队列的三种形式，分别是单向队列、双向队列以及优先级队列。其实大家听名字也可以听得出来他们之间的区别，单向队列遵循先进先出的原则，而且一端只能插入，另一端只能删除。双向队列则两端都可插入和删除，如果限制双向队列的某一段的方法，则可以达到和单向队列同样的功能。最后优先级队列，则是在插入元素的时候进行了优先级别排序，在实际应用中单项队列和优先级队列使用的比较多。后面讲解了堆这种数据结构，我们会用堆来实现优先级队列，改善优先级队列插入元素的时间。

　　通过前面讲的栈以及本篇讲的队列这两种数据结构，我们稍微总结一下：

　　①、栈、队列（单向队列）、优先级队列通常是用来简化某些程序操作的数据结构，而不是主要作为存储数据的。

　　②、在这些数据结构中，只有一个数据项可以被访问。

　　③、栈允许在栈顶压入（插入）数据，在栈顶弹出（移除）数据，但是只能访问最后一个插入的数据项，也就是栈顶元素。

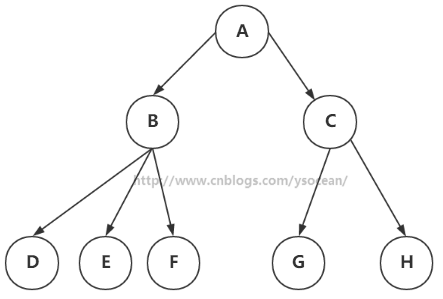
　　④、队列（单向队列）只能在队尾插入数据，对头删除数据，并且只能访问对头的数据。而且队列还可以实现循环队列，它基于数组，数组下标可以从数组末端绕回到数组的开始位置。

　　⑤、优先级队列是有序的插入数据，并且只能访问当前元素中优先级别最大（或最小）的元素。

　　⑥、这些数据结构都能由数组实现，但是可以用别的机制（后面讲的链表、堆等数据结构）实现。

### 树

**树**（tree）是一种抽象数据类型（ADT），用来模拟具有树状结构性质的数据集合。它是由n（n>0）个有限**节点**通过连接它们的**边**组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。

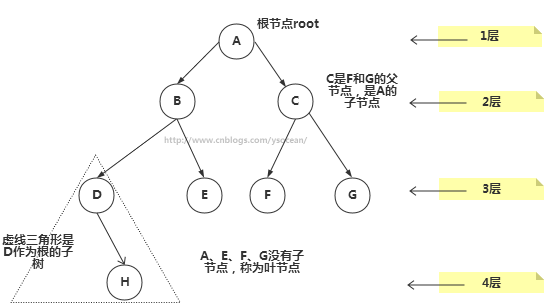


　　①、节点：上图的圆圈，比如A,B,C等都是表示节点。节点一般代表一些实体，在java面向对象编程中，节点一般代表对象。

　　②、边：连接节点的线称为边，边表示节点的关联关系。一般从一个节点到另一个节点的**唯一方法**就是沿着一条顺着有边的道路前进。在Java当中通常表示引用。

树有很多种，向上面的一个节点有多余两个的子节点的树，称为多路树，后面会讲解2-3-4树和外部存储都是多路树的例子。而每个节点最多只能有两个子节点的一种形式称为二叉树，这也是本篇博客讲解的重点。

树的常用术语



　　①、**路径**：顺着节点的边从一个节点走到另一个节点，所经过的节点的顺序排列就称为“路径”。

　　②、**根**：树顶端的节点称为根。一棵树只有一个根，如果要把一个节点和边的集合称为树，那么从根到其他任何一个节点都必须有且只有一条路径。A是根节点。

　　③、**父节点**：若一个节点含有子节点，则这个节点称为其子节点的父节点；B是D的父节点。

　　④、**子节点**：一个节点含有的子树的根节点称为该节点的子节点；D是B的子节点。

　　⑤、**兄弟节点**：具有相同父节点的节点互称为兄弟节点；比如上图的D和E就互称为兄弟节点。

　　⑥、**叶节点**：没有子节点的节点称为叶节点，也叫叶子节点，比如上图的A、E、F、G都是叶子节点。

　　⑦、**子树**：每个节点都可以作为子树的根，它和它所有的子节点、子节点的子节点等都包含在子树中。

　　⑧、**节点的层次**：从根开始定义，根为第一层，根的子节点为第二层，以此类推。

　　⑨、**深度**：对于任意节点n,n的深度为从根到n的唯一路径长，根的深度为0；

　　⑩、**高度**：对于任意节点n,n的高度为从n到一片树叶的最长路径长，所有树叶的高度为0；

#### 二叉树

　　二叉树：树的每个节点最多只能有两个子节点

　　上图的第一幅图B节点有DEF三个子节点，就不是二叉树，称为多路树；而第二幅图每个节点最多只有两个节点，是二叉树，并且二叉树的子节点称为“左子节点”和“右子节点”。上图的D,E分别是B的左子节点和右子节点。

　　如果我们给二叉树加一个额外的条件，就可以得到一种被称作二叉搜索树(binary search tree)的特殊二叉树。

　　二叉搜索树要求：若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉排序树。



##### 查找节点

查找某个节点，我们必须从根节点开始遍历。

　　①、查找值比当前节点值大，则搜索右子树；

　　②、查找值等于当前节点值，停止搜索（终止条件）；

　　③、查找值小于当前节点值，则搜索左子树；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | //查找节点  public Node find(int key) {      Node current = root;      while(current != null){          if(current.data > key){//当前值比查找值大，搜索左子树              current = current.leftChild;          }else if(current.data < key){//当前值比查找值小，搜索右子树              current = current.rightChild;          }else{              return current;          }      }      return null;//遍历完整个树没找到，返回null  } |

　　用变量current来保存当前查找的节点，参数key是要查找的值，刚开始查找将根节点赋值到current。接在在while循环中，将要查找的值和current保存的节点进行对比。如果key小于当前节点，则搜索当前节点的左子节点，如果大于，则搜索右子节点，如果等于，则直接返回节点信息。当整个树遍历完全，即current == null，那么说明没找到查找值，返回null。

**树的效率**：查找节点的时间取决于这个节点所在的层数，每一层最多有2n-1个节点，总共N层共有2n-1个节点，那么时间复杂度为O(logn),底数为2。

##### 插入节点

　　 要插入节点，必须先找到插入的位置。与查找操作相似，由于二叉搜索树的特殊性，待插入的节点也需要从根节点开始进行比较，小于根节点则与根节点左子树比较，反之则与右子树比较，直到左子树为空或右子树为空，则插入到相应为空的位置，在比较的过程中要注意保存父节点的信息 及 待插入的位置是父节点的左子树还是右子树，才能插入到正确的位置。

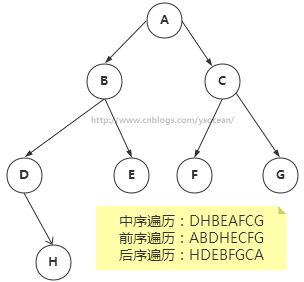
##### 遍历树

遍历树是根据一种特定的顺序访问树的每一个节点。比较常用的有前序遍历，中序遍历和后序遍历。而二叉搜索树最常用的是中序遍历。

**①、中序遍历:左子树——》根节点——》右子树**

**②、前序遍历:根节点——》左子树——》右子树**

**③、后序遍历:左子树——》右子树——》根节点**

****

##### 查找最大值和最小值

这没什么好说的，要找最小值，先找根的左节点，然后一直找这个左节点的左节点，直到找到没有左节点的节点，那么这个节点就是最小值。同理要找最大值，一直找根节点的右节点，直到没有右节点，则就是最大值。

##### 删除节点

删除节点是二叉搜索树中最复杂的操作，删除的节点有三种情况，前两种比较简单，但是第三种却很复杂。

　　1、该节点是叶节点（没有子节点）

　　2、该节点有一个子节点

　　3、该节点有两个子节点

　　下面我们分别对这三种情况进行讲解。

##### 二叉树的效率

#### 红黑树

### 堆

### 图

## 算法

### 排序

#### O(n^2)

下面讲的三种排序，冒泡、选择、插入用大 O 表示法都需要 O(N2) 时间级别。一般不会选择冒泡排序，虽然冒泡排序书写是最简单的，但是平均性能是没有选择排序和插入排序好的。

　　选择排序把交换次数降低到最低，但是比较次数还是挺大的。当数据量小，并且交换数据相对于比较数据更加耗时的情况下，可以应用选择排序。

在大多数情况下，假设数据量比较小或基本有序时，插入排序是三种算法中最好的选择。

后面我们会讲解高级排序，大O表示法的时间级别将比O(N2)小。

##### 冒泡排序

这个名词的由来很好理解，一般河水中的冒泡，水底刚冒出来的时候是比较小的，随着慢慢向水面浮起会逐渐增大，这物理规律我不作过多解释，大家只需要了解即可。

　　冒泡算法的运作规律如下：

　　①、比较相邻的元素。如果第一个比第二个大，就交换他们两个。

　　②、对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数（也就是第一波冒泡完成）。

　　③、针对所有的元素重复以上的步骤，除了最后一个。

　　④、持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。



**冒泡排序解释：**

　　冒泡排序是由两个for循环构成，第一个for循环的变量 i 表示总共需要多少轮比较，第二个for循环的变量 j 表示每轮参与比较的元素下标【0,1，......，length-i】，因为每轮比较都会出现一个最大值放在最右边，所以每轮比较后的元素个数都会少一个，这也是为什么 j 的范围是逐渐减小的。相信大家理解之后快速写出一个冒泡排序并不难。

**冒泡排序性能分析：**

假设参与比较的数组元素个数为 N，则第一轮排序有 N-1 次比较，第二轮有 N-2 次，如此类推，这种序列的求和公式为：

　　（N-1）+（N-2）+...+1 = N\*（N-1）/2

　　当 N 的值很大时，算法比较次数约为 N2/2次比较，忽略减1。

　　假设数据是随机的，那么每次比较可能要交换位置，可能不会交换，假设概率为50%，那么交换次数为 N2/4。不过如果是最坏的情况，初始数据是逆序的，那么每次比较都要交换位置。

　　交换和比较次数都和N2 成正比。由于常数不算大 O 表示法中，忽略 2 和 4，那么冒泡排序运行都需要 O(N2) 时间级别。

　　其实无论何时，只要看见一个循环嵌套在另一个循环中，我们都可以怀疑这个算法的运行时间为 O(N2)级，外层循环执行 N 次，内层循环对每一次外层循环都执行N次（或者几分之N次）。这就意味着大约需要执行N2次某个基本操作。

##### 选择排序

选择排序可以说是最简单的一种排序方法：

1.找到数组中最小的那个元素

2.将最小的这个元素和数组中第一个元素交换位置

3.在剩下的元素中找到最小的的元素，与数组第二个元素交换位置

重复以上步骤，即可以得到有序数组。



|  |
| --- |
| **public** **static** **int**[] sort(**int**[] src){  **if**(src == **null** || src.length == 0){  **return** src;  }    **int** min = 0;  **for**(**int** i=0; i<src.length; i++){  min = i;  **for**(**int** j = i+1; j<src.length; j++){  //每次遍历比较，找出最小（或者最大）下标  **if**(src[min] > src[j]){  min = j;  }  }    //如果遍历找到比第一位更小的数，调换位置  **if**(i != min){  **int** tem = src[i];  src[i] = src[min];  src[min] = tem;  }  }    **return** src;  } |

效率：对于长度为N的数组，选择排序需要大约N²/2次比较和N次交换。也即最好、最差、平均时间效率均为O（n²），只需要一个辅助变量帮助交换元素。

选择排序可以看成是冒泡排序的扩展，一个是把最小或最大的选出来，再交换，一个是一直交换直到最大最小的出现在正确的位置上，选择排序相对于冒泡排序，比较次数是一样的，但是交换次数要少很多。

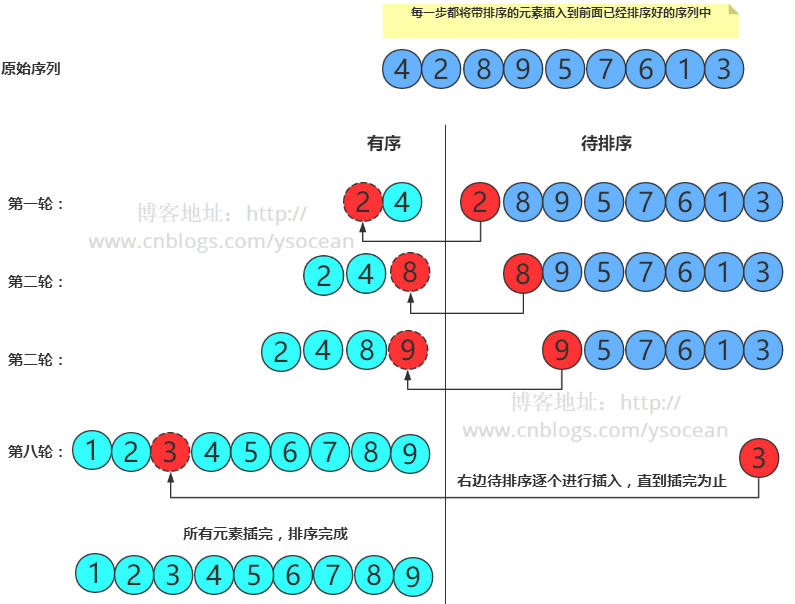
##### 插入排序

直接插入排序基本思想是每一步将一个待排序的记录，插入到前面已经排好序的有序序列中去，直到插完所有元素为止。

插入排序类似整理扑克牌，将每一张牌插到其他已经有序的牌中适当的位置。

插入排序由N-1趟排序组成，对于P=1到N-1趟，插入排序保证从位置0到位置P上的元素为已排序状态。

简单的说，就是插入排序总共需要排序N-1趟，从index为1开始，讲该位置上的元素与之前的元素比较，放入合适的位置，这样循环下来之后，即为有序数组。



|  |
| --- |
| **public** **static** **int**[] sort(**int**[] src){  **if**(src == **null** || src.length == 0){  **return** src;  }    **for**(**int** i=1; i<src.length; i++){  **for**(**int** j=0; j<i; j++){  **if**(src[i] < src[j]){  **int** temp = src[i];  **for**(**int** k=i; k>j; ){  src[k] = src[--k];  }  src[j] = temp;  }  }  }    **return** src;  } |

效率：如果目标是把n个元素的序列升序排列，那么采用插入排序存在最好情况和最坏情况。最好情况就是，序列已经是升序排列了，在这种情况下，需要进行的比较操作需（n-1）次即可。最坏情况就是，序列是降序排列，那么此时需要进行的比较共有n(n-1)/2次。插入排序的赋值操作是比较操作的次数加上 (n-1）次。平均来说插入排序算法的时间复杂度为O(n^2）

#### 希尔排序

把记录按步长 gap 分组，对每组记录采用直接插入排序方法进行排序。

随着步长逐渐减小，所分成的组包含的记录越来越多，当步长的值减小到 1 时，整个数据合成为一组，构成一组有序记录，则完成排序。

|  |
| --- |
| **public** **static** **int**[] sort(**int**[] a){  Integer h = a.length;  Integer temp = 0;  **while**(h >= 1) {  **for**(**int** i=h;i<a.length;i++) {  **for**(**int** j=i;j>=h && a[j] < a[j-h];j -= h) {  temp = a[j];  a[j] = a[j-h];  a[j-h] = temp;    }  }  h /= 9;  }  **return** a;  } |

#### 堆排序

#### 归并排序

#### 快速排序

#### 桶式排序

### 查找

### 递归

https://www.cnblogs.com/ysocean/p/8005694.html

递归，就是在运行的过程中调用自己。

　　递归必须要有三个要素：

　　①、边界条件

　　②、递归前进段

　　③、递归返回段

　　当边界条件不满足时，递归前进；当边界条件满足时，递归返回。

#### 求一个数的阶乘

|  |
| --- |
| n! = n\*(n-1)\*(n-2)\*......1 |

**规定：**

**①、0！=1**

**②、1！=1**

**③、负数没有阶乘**

#### 递归二分查找

注意：二分查找的数组一定是有序的！！！

　　在有序数组array[]中，不断将数组的中间值（mid）和被查找的值比较，如果被查找的值等于array[mid],就返回下标mid; 否则，就将查找范围缩小一半。如果被查找的值小于array[mid], 就继续在左半边查找;如果被查找的值大于array[mid],  就继续在右半边查找。 直到查找到该值或者查找范围为空时， 查找结束。

#### 分治算法

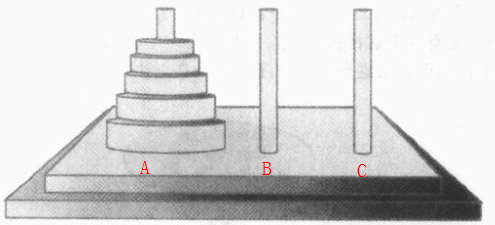
当我们求解某些问题时，由于这些问题要处理的数据相当多，或求解过程相当复杂，使得直接求解法在时间上相当长，或者根本无法直接求出。对于这类问题，我们往往先把它分解成几个子问题，找到求出这几个子问题的解法后，再找到合适的方法，把它们组合成求整个问题的解法。如果这些子问题还较大，难以解决，可以再把它们分成几个更小的子问题，以此类推，直至可以直接求出解为止。这就是分治策略的基本思想。

　　上面讲的递归的二分查找法就是一个分治算法的典型例子，分治算法常常是一个方法，在这个方法中含有两个对自身的递归调用，分别对应于问题的两个部分。

　　二分查找中，将查找范围分成比查找值大的一部分和比查找值小的一部分，每次递归调用只会有一个部分执行。

#### 汉诺塔问题

汉诺塔问题是由很多放置在三个塔座上的盘子组成的一个古老的难题。如下图所示，所有盘子的直径是不同的，并且盘子中央都有一个洞使得它们刚好可以放在塔座上。所有的盘子刚开始都放置在A 塔座上。这个难题的目标是将所有的盘子都从塔座A移动到塔座C上，每次只可以移动一个盘子，并且任何一个盘子都不可以放置在比自己小的盘子之上。



　　试想一下，如果只有两个盘子，盘子从小到大我们以数字命名（也可以想象为直径），两个盘子上面就是盘子1，下面是盘子2，那么我们只需要将盘子1先移动到B塔座上，然后将盘子2移动到C塔座，最后将盘子1移动到C塔座上。即完成2个盘子从A到C的移动。

　　如果有三个盘子，那么我们将盘子1放到C塔座，盘子2放到B塔座，在将C塔座的盘子1放到B塔座上，然后将A塔座的盘子3放到C塔座上，然后将B塔座的盘子1放到A塔座，将B塔座的盘子2放到C塔座，最后将A塔座的盘子1放到C塔座上。

　　如果有四个，五个，N个盘子，那么我们应该怎么去做？这时候递归的思想就很好解决这样的问题了，当只有两个盘子的时候，我们只需要将B塔座作为中介，将盘子1先放到中介塔座B上，然后将盘子2放到目标塔座C上，最后将中介塔座B上的盘子放到目标塔座C上即可。

**所以无论有多少个盘子，我们都将其看做只有两个盘子。假设有 N 个盘子在塔座A上，我们将其看为两个盘子，其中(N-1)~1个盘子看成是一个盘子，最下面第N个盘子看成是一个盘子，那么解决办法为：**

**①、先将A塔座的第(N-1)~1个盘子看成是一个盘子，放到中介塔座B上，然后将第N个盘子放到目标塔座C上。**

**②、然后A塔座为空，看成是中介塔座，B塔座这时候有N-1个盘子，将第(N-2)~1个盘子看成是一个盘子，放到中介塔座A上，然后将B塔座的第(N-1)号盘子放到目标塔座C上。**

**③、这时候A塔座上有(N-2)个盘子，B塔座为空，又将B塔座视为中介塔座，重复①，②步骤，直到所有盘子都放到目标塔座C上结束。**

　　简单来说，跟把大象放进冰箱的步骤一样，递归算法为：

　　①、从初始塔座A上移动包含n-1个盘子到中介塔座B上。

　　②、将初始塔座A上剩余的一个盘子（最大的一个盘子）放到目标塔座C上。

　　③、将中介塔座B上n-1个盘子移动到目标塔座C上。

|  |
| --- |
| **package** org.xzh.springboot.sort;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** HannuotaMain {    **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(HannuotaMain.**class**);  **public** **static** **void** move(**int** disk, String from, String temp, String to){  **if**(disk == 1){  ***logger***.info("将盘子{}从塔座{}移到塔座{}", disk, from, to);  }**else**{  *move*(disk-1, from, to, temp);  ***logger***.info("将盘子{}从塔座{}移到塔座{}", disk, from, to);  *move*(disk-1, temp, from, to);  }  };    **public** **static** **void** main(String[] args) {  *move*(4, "A", "B", "c");  }  } |

#### 归并排序

# 设计模式（23种）

## 创建型模式

### 工厂方法模式

### 抽象工厂模式

### 单例模式

### 创建者模式

### 原型模式

## 结构型模式

### 适配器模式

### 装饰器模式

### 代理模式

### 外观模式

### 桥接模式

### 组合模式

### 亨元模式

## 行为型模式

### 策略模式

### 模板方法模式

### 观察者模式

### 迭代子模式

### 责任链模式

### 命令模式

### 备忘录模式

### 状态模式

### 访问者模式

### 终结者模式

### 解释器模式

# 常用框架

## Springboot

1、属于spring旗下的一个项目，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，它使用“习惯优于配置”的理念，从而使开发人员不再需要定义样板化的配置。使用springboot很容易创建一个独立运行的spring项目，并且可以几乎不使用spring配置或者使用很少的配置。

2、结构

1）、springboot是一个maven项目，只需要在pom.xml添加依赖就可以。

2）、在pom.xml添加一个parent依赖，控制整个springboot项目的版本

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.6.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

3）、spring提供了很多spring-boot-starter-XXX的maven依赖，来简化maven依赖配置

比如：web相关spring-boot-starter-web，会自动关联一些常见web依赖，比如springMVC， servlet、tomcat容器。。。因此无需在tomcat服务器上部署。

比如：spring-boot-starter-thymeleaf，会自动关联一些与thymeleaf相关的依赖

4）、编写springboot的入口类，一般命名是xxxApplication.java，其他的类都在它所在的目录或者子目录中，这样spring容器会自动扫描有相应注解的类。入口类上的注解是SpringBootApplicaton,类中有个Main方法作为程序入口，springboot框架提供一个方法启动项目Application.run(本类名.class，args)。

5）、springboot启动时会自动扫描src/main/resources中配置application.properties

6）、入口注解@SpringBootApplication注解是一个组合注解，它包含@configuration、@EnableAutoConfiguration和@componentScan三个注解组成。它的核心是@EnableAutoConfiguration注解。这个注解中有一个@Import注解，它导入配置功能EnableAutoConfigurationImportSelector，这个类有个方法扫描具有/META-INF/spring.factories文件的jar。spring.factories中声明了哪些自动配置。然后我们在application.properties文件中进行相应的配置，比如数据库连接，使springboot中的自动配置中的参数是我们需要的配置。

7）如果项目最后要以jar形式运行，必须加入spring-boot-maven-plugin插件

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<fork>true</fork>

</configuration>

</plugin>

8）@SpringBootApplication标签

@ EnableAutoConfiguration：启用Spring Boot的自动配置机制

@ ComponentScan：在应用程序所在的包上启用@Component 扫描

@Configuration：允许在上下文中注册额外的bean或导入额外的配置类

## Spring

Spring是一个开源框架，Spring是于2003 年兴起的一个轻量级的Java 开发框架，由Rod Johnson 在其著作Expert One-On-One J2EE Development and Design中阐述的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许使用者选择使用哪一个组件，同时为 J2EE 应用程序开发提供集成的框架。Spring使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。Spring的核心是控制反转（IoC）和面向切面（AOP）。简单来说，Spring是一个分层的JavaSE/EE full-stack(一站式) 轻量级开源框架。

**Spring的优点：**

1、方便解耦，简化开发 （高内聚低耦合）

Spring就是一个大工厂（容器），可以将所有对象创建和依赖关系维护，交给Spring管理

spring工厂是用于生成bean

2、AOP编程的支持

Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能

3、声明式事务的支持

只需要通过配置就可以完成对事务的管理，而无需手动编程

4、方便程序的测试

Spring对Junit4支持，可以通过注解方便的测试Spring程序

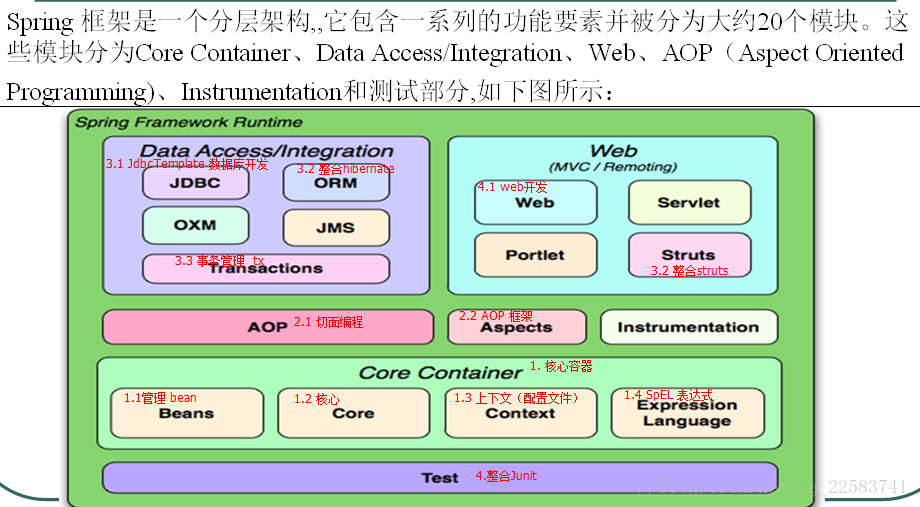
5、方便集成各种优秀框架

Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz等）的直接支持

6、降低JavaEE API的使用难度

Spring 对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低

**Spring体系：**



### IOC-依赖注入（反转控制）

通过配置文件获取对象

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd">  <!-- 配置service  <bean> 配置需要创建的对象  id ：用于之后从spring容器获得实例时使用的  class ：需要创建实例的全限定类名  -->  <bean id="userServiceId" class="com.itheima.a\_ioc.UserServiceImpl"></bean>  </beans> |

|  |
| --- |
| //获取容器  String xmlPath = "com/itheima/a\_ioc/beans.xml";  ApplicationContext applicationContext = new ClassPathXmlApplicationContext(xmlPath);  //2获得内容 --不需要自己new，都是从spring容器获得  UserService userService = (UserService) applicationContext.getBean("userServiceId");  userService.addUser(); |

**基于XML属性依赖注入**

* 依赖注入方式：手动装配 和 自动装配   
  + 手动装配：一般进行配置信息都采用手动   
    基于xml装配：构造方法、setter方法
  + 基于注解装配：

集合依赖注入

|  |
| --- |
| 集合的注入都是给<property>添加子标签  数组：<array>  List：<list>  Set：<set>  Map：<map> ，map存放k/v 键值对，使用<entry>描述  Properties：<props> <prop key=""></prop> 【】  普通数据：<value>  引用数据：<ref> |

**基于注解的依赖注入**

|  |
| --- |
| //注解使用前提，添加命名空间，让spring扫描含有注解类  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd">  <!-- 组件扫描，扫描含有注解的类 -->  **<context:component-scan base-package="com.itheima.g\_annotation.a\_ioc"></context:component-scan>**  </beans> |

|  |
| --- |
| 开发中：使用注解 取代 xml配置文件。  1.**@Component**取代<bean class="">  @Component("id") 取代 <bean id="" class="">  2.web开发，提供3个@Component注解衍生注解（功能一样）取代  **@Repository ：dao层**  **@Service：service层**  **@Controller：web层**  3.依赖注入，给私有字段设值，也可以给setter方法设值  普通值：**@Value(" ")**  引用值：  方式1：按照【类型】注入  **@Autowired**  方式2：按照【名称】注入1  **@Autowired**  **@Qualifier("名称")**  方式3：按照【名称】注入2  **@Resource("名称")**  4.生命周期  **初始化：@PostConstruct**  **销毁：@PreDestroy**  5.作用域  **@Scope("prototype") 多例** |

### AOP-面向切面编程

在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP（面向对象编程）的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

AOP采取横向抽取机制，取代了传统纵向继承体系重复性代码

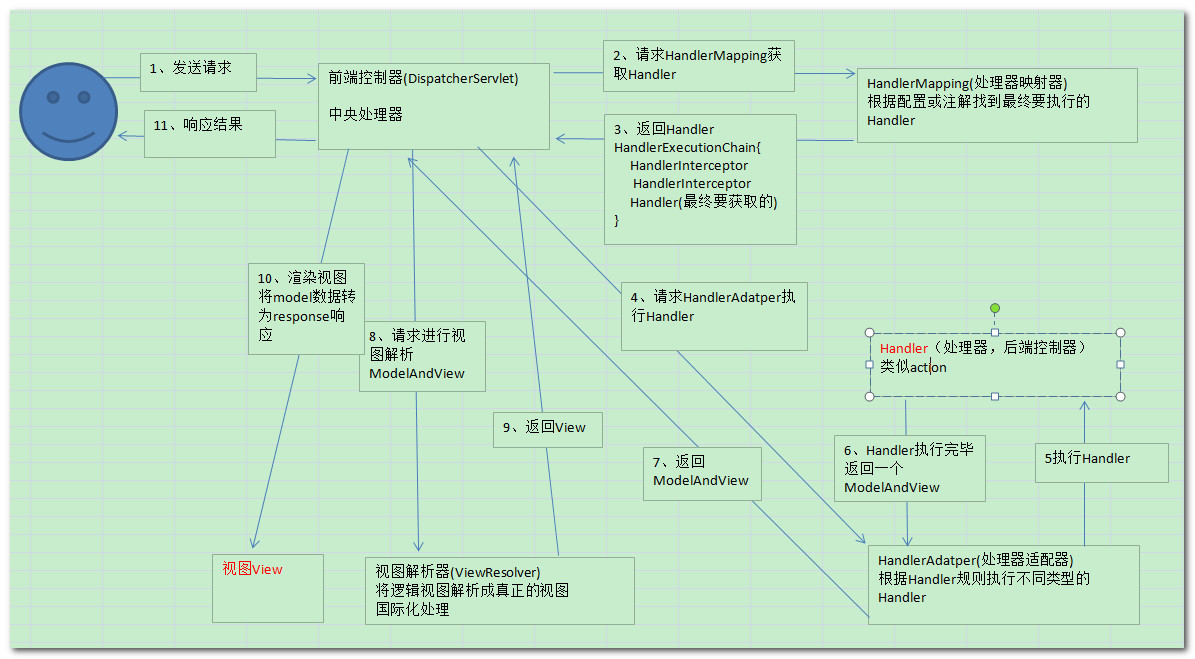
经典应用：事务管理、性能监视、安全检查、缓存 、日志等

Spring AOP使用纯Java实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类织入增强代码

AspectJ是一个基于Java语言的AOP框架，Spring2.0开始，Spring AOP引入对Aspect的支持，AspectJ扩展了Java语言，提供了一个专门的编译器，在编译时提供横向代码的织入

https://blog.csdn.net/qq\_22583741/article/details/79589910

## Spring mvc



### SpringMVC流程

1、  用户发送请求至前端控制器DispatcherServlet。

2、  DispatcherServlet收到请求调用HandlerMapping处理器映射器。

3、  处理器映射器找到具体的处理器(可以根据xml配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

4、  DispatcherServlet调用HandlerAdapter处理器适配器。

5、  HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。

6、  Controller执行完成返回ModelAndView。

7、  HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet。

8、  DispatcherServlet将ModelAndView传给ViewReslover视图解析器。

9、  ViewReslover解析后返回具体View。

10、DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。

11、 DispatcherServlet响应用户。

### 组件说明

以下组件通常使用框架提供实现：

DispatcherServlet：作为前端控制器，整个流程控制的中心，控制其它组件执行，统一调度，降低组件之间的耦合性，提高每个组件的扩展性。

HandlerMapping：通过扩展处理器映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

HandlAdapter：通过扩展处理器适配器，支持更多类型的处理器。

ViewResolver：通过扩展视图解析器，支持更多类型的视图解析，例如：jsp、freemarker、pdf、excel等。

**组件：**  
**1、前端控制器DispatcherServlet（不需要工程师开发）,由框架提供**  
作用：接收请求，响应结果，相当于转发器，中央处理器。有了dispatcherServlet减少了其它组件之间的耦合度。  
用户请求到达前端控制器，它就相当于mvc模式中的c，dispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet的存在降低了组件之间的耦合性。

**2、处理器映射器HandlerMapping(不需要工程师开发),由框架提供**  
作用：根据请求的url查找Handler  
HandlerMapping负责根据用户请求找到Handler即处理器，springmvc提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

**3、处理器适配器HandlerAdapter**  
作用：按照特定规则（HandlerAdapter要求的规则）去执行Handler  
通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

**4、处理器Handler(需要工程师开发)**  
**注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler**  
Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。  
由于Handler涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发Handler。

**5、视图解析器View resolver(不需要工程师开发),由框架提供**  
作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）  
View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。 springmvc框架提供了很多的View视图类型，包括：jstlView、freemarkerView、pdfView等。  
一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

**6、视图View(需要工程师开发jsp...)**  
View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf...）

**核心架构的具体流程步骤如下：**  
1、首先用户发送请求——>DispatcherServlet，前端控制器收到请求后自己不进行处理，而是委托给其他的解析器进行处理，作为统一访问点，进行全局的流程控制；  
2、DispatcherServlet——>HandlerMapping， HandlerMapping 将会把请求映射为HandlerExecutionChain 对象（包含一个Handler 处理器（页面控制器）对象、多个HandlerInterceptor 拦截器）对象，通过这种策略模式，很容易添加新的映射策略；  
3、DispatcherServlet——>HandlerAdapter，HandlerAdapter 将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；  
4、HandlerAdapter——>处理器功能处理方法的调用，HandlerAdapter 将会根据适配的结果调用真正的处理器的功能处理方法，完成功能处理；并返回一个ModelAndView 对象（包含模型数据、逻辑视图名）；  
5、ModelAndView的逻辑视图名——> ViewResolver， ViewResolver 将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；  
6、View——>渲染，View会根据传进来的Model模型数据进行渲染，此处的Model实际是一个Map数据结构，因此很容易支持其他视图技术；  
7、返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。

下边两个组件通常情况下需要开发：

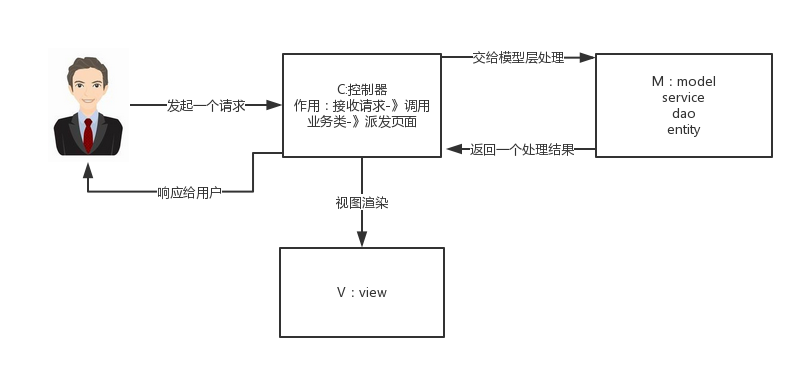
Handler：处理器，即后端控制器用controller表示。

View：视图，即展示给用户的界面，视图中通常需要标签语言展示模型数据。

### MVC

MVC：MVC是一种设计模式

MVC的原理图：



**分析：**

M-Model 模型（完成业务逻辑：有javaBean构成，service+dao+entity）

V-View 视图（做界面的展示  jsp，html……）

C-Controller 控制器（接收请求—>调用模型—>根据结果派发页面）

**springMVC是什么：**

springMVC是一个MVC的开源框架，springMVC=struts2+spring，springMVC就相当于是Struts2加上sring的整合，但是这里有一个疑惑就是，springMVC和spring是什么样的关系呢？这个在百度百科上有一个很好的解释：意思是说，springMVC是spring的一个后续产品，其实就是spring在原有基础上，又提供了web应用的MVC模块，可以简单的把springMVC理解为是spring的一个模块（类似AOP，IOC这样的模块），网络上经常会说springMVC和spring无缝集成，其实springMVC就是spring的一个子模块，所以根本不需要同spring进行整合。

## Mybatis

## Quartz

## Junit

## FreeMarker

## Zookeeper

## Docket

## Netty

### BIO

在jdk1.3或之前，java只提供阻塞IO编程，java.io包下面。到jdk1.4时才提供非阻塞io编程，在java.nio下面。

阻塞BIO中，网络编程用java.net.Socket和java.net.ServerSocket。**socket通过TCP的三次握手建立连接、四次挥手释放连接**。每个客户端或者每次Socket请求，服务器都会建立一个线程来处理，处理完之后销毁线程，这种模式基本无法处理高并发量的请求。







#### Server

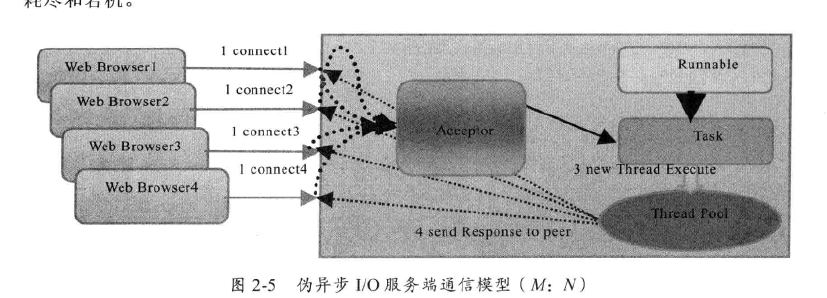
|  |
| --- |
| **package** org.xzh.springboot.io.bio;  **import** java.net.ServerSocket;  **import** java.net.Socket;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** Server {    **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(Server.**class**);  **private** **void** run() {  **int** port = 9900;  ServerSocket server = **null**;  **try**{  server = **new** ServerSocket(port);  **while**(**true**){  Socket socket = server.accept();  **new** Thread(**new** SocketHandler(socket)).start();  }  }**catch**(Exception e){    }  }    **public** **class** SocketHandler **implements** Runnable{    **private** Socket socket = **null**;    **public** SocketHandler(Socket socket){  **this**.socket = socket;  }  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName()+" : "+socket.getInetAddress().getHostAddress());  }    }    **public** **static** **void** main(String[] args) {  Server server = **new** Server();  server.run();  }  } |

#### Client

|  |
| --- |
| package org.xzh.springboot.io.bio;  import java.io.IOException;  import java.io.InputStream;  import java.io.OutputStream;  import java.net.Socket;  import java.net.UnknownHostException;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  public class Client {    private static final Logger logger = LoggerFactory.getLogger(Client.class);  public void run(){  Socket socket = null;  try {  socket = new Socket("127.0.0.1", 9900);  //InputStream is = socket.getInputStream();  OutputStream os = socket.getOutputStream();  os.write(Thread.currentThread().getName().getBytes("UTF-8"));    /\*byte[] bytes = new byte[1024];  is.read(bytes, 0, 1024);  logger.info(new String(bytes, "UTF-8"));\*/  } catch (UnknownHostException e) {  // TODO Auto-generated catch block  e.printStackTrace();  } catch (IOException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }finally{  if(socket != null){  try {  socket.close();  } catch (IOException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  }  } |

### 伪异步IO编程

基于传统阻塞IO，一客户端一线程的模式，有人优化该模型，在后端服务器引入线程池，所有客户端共享一定数量的线程，这样服务器的线程数量就不会随着客户端的增加而增加，防止海量并发导致线程资源耗尽。



### NIO

与阻塞IO的Socket和ServerSocket对应、非阻塞NIO引入java.nio.channes.SocketChannel和java.nio.channels.ServerSocketChannel。新增两个嵌套字都支持阻塞和非阻塞。

1、缓冲区（Buffer）

传统阻塞IO是面向流的IO，可以将数据直接写入和将数据直接读到stream对象。

非阻塞NIO，所有数据都是用缓冲区来处理。读取数据时，将直接读取到缓冲区中数据，写入数据时，将直接把数据写到缓冲区中。

缓冲区实质是是一个数组，通常是一个字节数组（ByteBuffer），也可以是其他缓冲区。但缓冲区不仅仅是一个数据，还提供了操作缓冲区的数据结构。基本类型中，处理Boolean之外，其他类型都有提供了对应的缓冲区。



2、通道（Channel）

Channel就是一个通道，网络数据通过channel读取和写入。通道（Channel）与流（Stream）不同之处在于通道是双向，流只能在一个方向上移动（一个流必须是InputStream或者OutputStream的子类），通道可以用于读、写或者同时进行。

channel是全双工。

channel可以分为两大类，用于网络读写的SelectableChannel和文件读写FileChannel。

3、多路复用器（Selector）

Selector会不断的轮询注册在其上的Channel，如果每个Channel上面发现了读或写事件，这个Channel就处于就绪状态，会被selector轮询出来，然后通过selectionKey可以或读取就绪的Channel集合，进行后续IO操作。

一个多路复用器Selector可以同时轮询多个Channel，由于JDK使用epoll()代替传统的select实现，所以它并没有最大连接句柄1024/2018的限制。这也就意味着只需要一个线程负责Selector的轮询，就可以接入成千上万的客户端。

#### Server

|  |
| --- |
| **package** org.xzh.springboot.io.nio;  **public** **class** TimerServer {  **public** **static** **void** main(String[] args) {  **int** port = 8080;    MultiplexerTimerServer mts = **new** MultiplexerTimerServer(port);  **new** Thread(mts).start();  }  } |

|  |
| --- |
| package org.xzh.springboot.io.nio;  import java.io.IOException;  import java.net.InetSocketAddress;  import java.nio.ByteBuffer;  import java.nio.channels.SelectionKey;  import java.nio.channels.Selector;  import java.nio.channels.ServerSocketChannel;  import java.nio.channels.SocketChannel;  import java.util.Date;  import java.util.Iterator;  import java.util.Set;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  /\*\*  \* 使用java.nio实现Socket网络通信  \* 这种非阻塞：需要考虑“读半包”、“写半包”的问题，本例子没有考虑这些问题  \*  \*/  public class MultiplexerTimerServer implements Runnable {    private static final Logger logger = LoggerFactory.getLogger(MultiplexerTimerServer.class);    /\*\*  \* 多路复用器  \*/  private Selector selector;    /\*\*  \* ServerSocket的Chanel类型  \*/  private ServerSocketChannel serverSocketChannel;    private volatile boolean stop;  public MultiplexerTimerServer(int port) {  try{  //打开selector  selector = Selector.open();  //打开ServerSocketChannel  serverSocketChannel = ServerSocketChannel.open();  //设置为非阻塞模式  serverSocketChannel.configureBlocking(false);  //绑定端口  serverSocketChannel.socket().bind(new InetSocketAddress(port), 1024);  //ServerSocketChannel注册到selector中  serverSocketChannel.register(selector, SelectionKey.OP\_ACCEPT);  }catch(Exception e){  logger.error("start serverSocketChannel is error, ",e);  System.exit(0);  }  }    public void stop(){  this.stop = true;  }  @Override  public void run() {  while(!stop){  try{  //设置休眠时间为1s  selector.select(1000);  //获取就绪状态的channel  Set<SelectionKey> selectionKeys = selector.selectedKeys();  Iterator<SelectionKey> it = selectionKeys.iterator();  SelectionKey key = null;  while(it.hasNext()){  key = it.next();  it.remove();  try{  handleInput(key);  }catch(Exception e){  key.cancel();  if(key.channel() != null){  key.channel().close();  }  }  }  }catch(Exception e){    }  }  }    private void handleInput(SelectionKey key) throws IOException{  if(key.isValid()){  if(key.isAcceptable()){ //判断网络类型为接入  ServerSocketChannel ssc = (ServerSocketChannel) key.channel();  //获取接入的SocketChannle  SocketChannel sc = ssc.accept();  //设置为非阻塞  sc.configureBlocking(false);  sc.register(selector, SelectionKey.OP\_READ);  }  if(key.isReadable()){ //判断网络类型为读写  SocketChannel sc = (SocketChannel) key.channel();  ByteBuffer byteBuffer = ByteBuffer.allocate(1024);  int readbyte = sc.read(byteBuffer);  if(readbyte > 0){  byteBuffer.flip();  byte[] bytes = new byte[byteBuffer.remaining()];  byteBuffer.get(bytes);  String body = new String(bytes, "UTF-8");  logger.info("the timer server reciver order:{}", body);  doWrite(sc, new Date().toString());  }else if(readbyte < 0){  //关闭链路  key.cancel();  sc.close();  }  }  }  }    private void doWrite(SocketChannel sc, String response) throws IOException {  byte[] bytes = response.getBytes("UTF-8");  ByteBuffer byteBuffer = ByteBuffer.allocate(bytes.length);  byteBuffer.put(bytes);  byteBuffer.flip();  sc.write(byteBuffer);  }  } |

#### Client

|  |
| --- |
| **package** org.xzh.springboot.io.nio;  **public** **class** TimerClient {  **public** **static** **void** main(String[] args) {  String host = "127.0.0.1";  **int** port = 8080;  **new** Thread(**new** TimerClientHandler(host, port)).start();  }  } |

|  |
| --- |
| package org.xzh.springboot.io.nio;  import java.io.IOException;  import java.net.InetSocketAddress;  import java.nio.ByteBuffer;  import java.nio.channels.ClosedChannelException;  import java.nio.channels.SelectionKey;  import java.nio.channels.Selector;  import java.nio.channels.SocketChannel;  import java.util.Iterator;  import java.util.Set;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  public class TimerClientHandler implements Runnable {    private static final Logger logger = LoggerFactory.getLogger(TimerClientHandler.class);    private String host;  private int port;    private Selector selector;  private SocketChannel socketChannel;  private volatile boolean stop;  public TimerClientHandler(String host, int port) {  super();  this.host = host;  this.port = port;  try{  selector = Selector.open();  socketChannel = SocketChannel.open();  socketChannel.configureBlocking(false);  }catch(Exception e){  logger.error("start timerClientHandler is error, ", e);  }  }  @Override  public void run() {  try{  doConnect();  }catch(IOException e){  logger.error("error-->", e);  System.exit(0);  }  while(!stop){  try{  selector.select(1000);  Set<SelectionKey> selectionKeys = selector.selectedKeys();  Iterator<SelectionKey> it = selectionKeys.iterator();  SelectionKey key = null;  while(it.hasNext()){  key = it.next();  it.remove();  try{  doHandler(key);  }catch(Exception e){  key.cancel();  if(key.channel() != null){  key.channel().close();  }  }  }  }catch(Exception e){  logger.error("error-->", e);  System.exit(1);  }  }    if(selector != null){  try {  selector.close();  } catch (IOException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }    private void doHandler(SelectionKey key) throws IOException {  if(key.isValid()){  SocketChannel sc = (SocketChannel) key.channel();  if(key.isConnectable()){  sc.register(selector, SelectionKey.OP\_READ);  doWrit(sc);  }else{  System.exit(1);  }    if(key.isReadable()){  ByteBuffer byteBuffer = ByteBuffer.allocate(1024);  int readBytes = sc.read(byteBuffer);  if(readBytes > 0){  byte[] bytes = new byte[readBytes];  byteBuffer.get(bytes);  byteBuffer.flip();  logger.info("Now is {}", new String(bytes, "UTF-8"));  this.stop = true;  }else if(readBytes < 0){  key.cancel();  sc.close();  }  }  }  }    private void doConnect() throws IOException {  if(socketChannel.connect(new InetSocketAddress(host, port))){  socketChannel.register(selector, SelectionKey.OP\_READ);  doWrit(socketChannel);  }else{  socketChannel.register(selector, SelectionKey.OP\_CONNECT);  }  }    private void doWrit(SocketChannel sc) throws IOException {  byte[] bytes = "QUERY TIME ORDER".getBytes("UTF-8");  ByteBuffer byteBuffer = ByteBuffer.allocate(bytes.length);  byteBuffer.put(bytes);  byteBuffer.flip();  sc.write(byteBuffer);  if(!byteBuffer.hasRemaining()){  logger.info("send order to server success");  }  }  } |

### Netty

作为一个NIO服务器，需要能处理网络的闪断、客户端重复接入，客户端的安全认证，信息的编解码，半包读写情况。

netty是业界最流行的NIO框架之一。很多业界主流框架（比如RPC框架dubbo）都用Netty来构建高性能的异步通信能力。

**1、TCP粘包/拆包**

无论是服务端还是客户端，当我们发送或者读取信息的时候，都要考虑TCP底层拆包/粘包机制。

TCP是个“流”协议，所谓流，就是没有界限的一串数据。TCP底层并不了解上层的业务数据的具体含义，它会根据TCP缓冲区的实际大小进行划分，所以业界认为，一个完整的包可能会被TCP拆成多个包进行发送，也有可能把多个小的包分装成大的数据包进行发送，这就是所谓TCP粘包/拆包的问题。

发生的原因：

（1）、应用程序write写入的字节大小大于嵌套字发送缓冲区的大小。

（2）、进行MSS大小的TCP分片

（3）、以太网帧的payload大于MTU进行的IP分片

解决方案：

（1）、信息定长（FixedLengthFrameDecoder）

（2）、在包尾添加分隔符

回车换行符（LineBasedFrameDecoder）

自定义分隔符（DelimiterBasedFrameDecoder）

（3）、信息分为消息头和信息体，消息头包含信息总长度或者消息体长度（LengthFieldBasedFrameDecoder，最常用）

（4）、更复杂的应用协议

利用LineBasedFrameDecoder解决TCP粘包问题：依次遍历ByteBuf的可读字节，判断是否有“\n”或者“\r\n”，如果有，就以此位置为结束位置，从可读索引到结束位置之间的字节组成一行。同时支持配置单行最大字节数，如果超出最大字节数还没有发现换行符，则抛出异常，同时忽略掉之前读到的异常码流，设置最大字节数是防止缺少分隔符导致内存异常。

利用StringDecoder将受到的对象转换成字符串。

**2、编解码**

主流RPC框架在远程调用服务时，很少直接使用java序列化进行信息的编解码和传输。因为java序列化有一些缺点：

（1）、无法跨语言

（2）、序列化后码流太大

（3）、序列化性能太低

业界主流的编解码框架：

（1）、Google的Protobuf：它将数据结构以.proto文件进行描述。很多RPC框架选用Protobuf作为编解码框架。

（2）、Facebook的Thrift：

（3）、JBoss Marshalling：

**dubbo的RPC框架使用hessian编解码**

**3、多协议栈开发**

Http协议（超文本传输协议）是建立在TCP传输协议上的应用协议。由于netty的http协议栈是基于netty的NIO通信框架开发，所以netty的http协议也是异步非阻塞。

Http协议的主要特点如下：

（1）、支持Client/Server模式

（2）、客户向服务器请求服务时，只需要指定服务URL，携带必要的请求参数或者信息体

（3）、Http允许传输任意类型的数据对象，传输的数据类型由Http消息头Content-Type加以标记。

（4）、无状态

Http请求由三部分组成：

（1）、Http请求行：请求行是一个方法符开头，以空格分割，后面跟着请求的URL和协议的版本，格式为：Method Request-URL HTTP-Version CRLF

（2）、Http消息头：允许客户端向服务器端传递请求的附加信息以及客户端自身的信息。



（3）、Http消息体：是可选，放请求信息。比如xml、json消息

Http协议的弊端

（1）、http协议为**半双工**协议。半双工协议是指数据可以在客户端与服务端两个方向上传输，但不能同时传输。它意味着在同一个时刻，只有一个方向上的数据传输。

（2）、Http消息数据大而繁琐。http请求包含http请求行，请求头，请求体。

（3）、针对服务器推送的黑客攻击。

Http基于TCP传输协议的应用协议，需要三次握手建立连接，四次挥手释放连接。

为了解决HTTP协议效率底下问题，HTML5定义了WebSocket协议，能更好的节省服务器资源和带宽并达到实时通讯。

WebSocket是HTML5开始提供一种浏览器与服务器之间进行**全双工**通讯的网络技术。

在WebSocket API中，浏览器与服务器只需要做**一次握手**的动作（TCP的握手与WebSocket握手是在不同层次），然后，浏览器与服务器之间就形成一条快速通道，两者就可以直接相互传递信息。WebSocket基于TCP双向全双工进行消息传递，同一时刻，既可以发送信息，也可以接受信息。

TCP的握手用来保证链接的建立，WebSocket的握手是在TCP链接建立后告诉服务器这是个WebSocket链接，服务器你要按WebSocket的协议来处理这个TCP链接。





## Mina

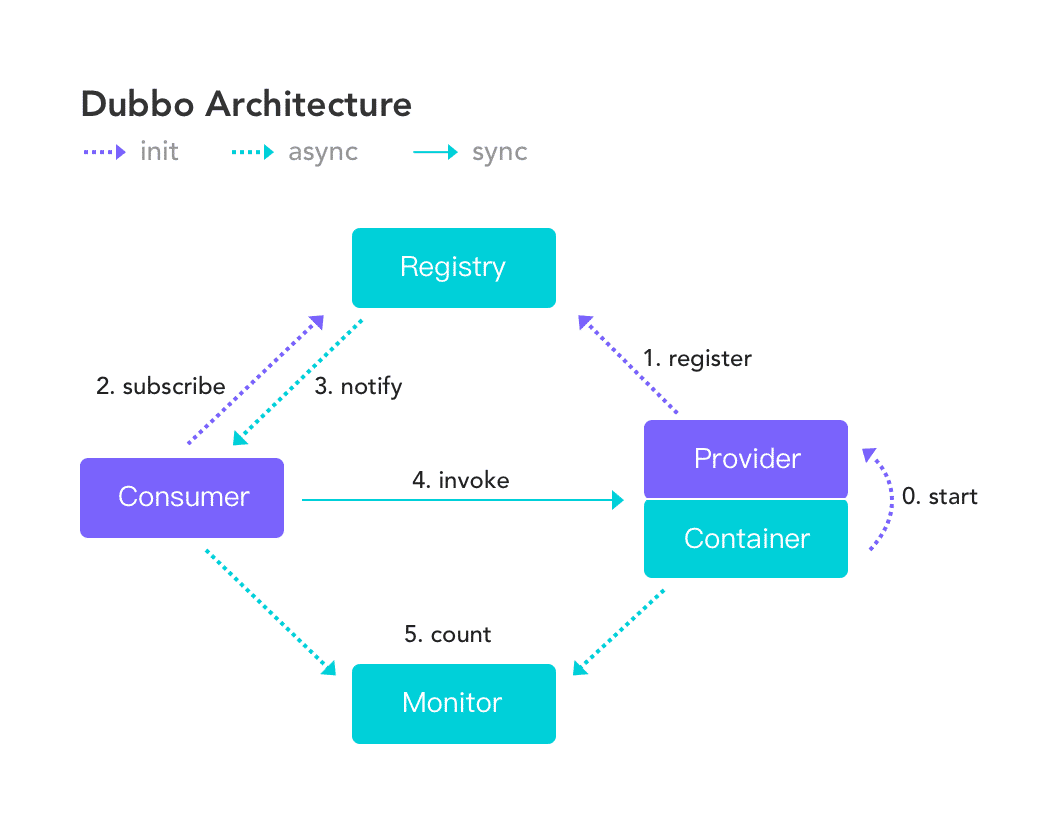
## Hadoop

# 分布式

## Rpc框架：dubbo

### 高性能java RPC框架

Apache Dubbo (incubating) |ˈdʌbəʊ| 是一款高性能、轻量级的开源Java RPC框架，它提供了三大核心能力：面向接口的远程方法调用，智能容错和负载均衡，以及服务自动注册和发现。



### 特性一览

#### 面向接口代理的高性能RPC调用

提供高性能的基于代理的远程调用能力，服务以接口为粒度，为开发者屏蔽远程调用底层细节。

#### 服务自动注册与发现

支持多种注册中心服务，服务实例上下线实时感知。

#### 智能负载均衡

内置多种负载均衡策略，智能感知下游节点健康状况，显著减少调用延迟，提高系统吞吐量。

#### 高度可扩展能力

遵循微内核+插件的设计原则，所有核心能力如Protocol、Transport、Serialization被设计为扩展点，平等对待内置实现和第三方实现。

#### 运行期流量调度

内置条件、脚本等路由策略，通过配置不同的路由规则，轻松实现灰度发布，同机房优先等功能。

#### 可视化的服务治理与运维

提供丰富服务治理、运维工具：随时查询服务元数据、服务健康状态及调用统计，实时下发路由策略、调整配置参数。

### 依赖

在springboot项目pom.xml添加如下内容

|  |
| --- |
| <dependency>  <groupId>com.alibaba.boot</groupId>  <artifactId>dubbo-spring-boot-starter</artifactId>  <version>0.2.0</version>  </dependency> |

### 配置

#### Xml配置

1、provider.xml示例

|  |
| --- |
| **<?**xml version="1.0" encoding="UTF-8"**?>**  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">  <dubbo:application name="hello-world-app" />  <dubbo:registry address="multicast://224.5.6.7:1234" />  <dubbo:protocol name="dubbo" port="20880" />  <dubbo:service interface="com.alibaba.dubbo.demo.DemoService" ref="demoServiceLocal" />  <dubbo:reference id="demoServiceRemote" interface="com.alibaba.dubbo.demo.DemoService" />  </beans> |

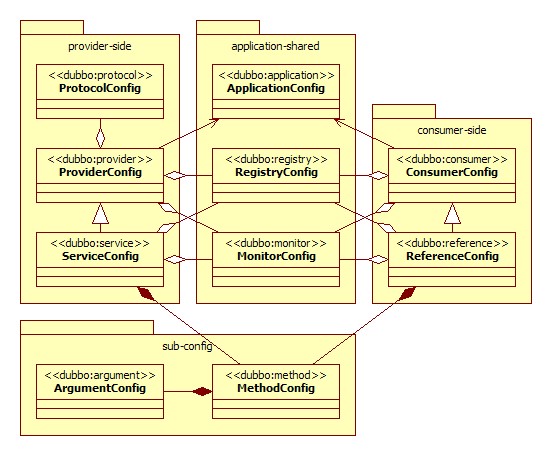
所有标签都支持自定义参数，用于不同扩展点实现的特殊配置，如：

|  |
| --- |
| <dubbo:protocol name="jms">  <dubbo:parameter key="queue" value="your\_queue" />  </dubbo:protocol> |

或：

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"  xmlns:p="http://www.springframework.org/schema/p"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">  <dubbo:protocol name="jms" p:queue="your\_queue" />  </beans> |

2、配置之间的关系



| **标签** | **用途** | **解释** |
| --- | --- | --- |
| <dubbo:service/> | 服务配置 | 用于暴露一个服务，定义服务的元信息，一个服务可以用多个协议暴露，一个服务也可以注册到多个注册中心 |
| <dubbo:reference/> | 引用配置 | 用于创建一个远程服务代理，一个引用可以指向多个注册中心 |
| <dubbo:protocol/> | 协议配置 | 用于配置提供服务的协议信息，协议由提供方指定，消费方被动接受 |
| <dubbo:application/> | 应用配置 | 用于配置当前应用信息，不管该应用是提供者还是消费者 |
| <dubbo:module/> | 模块配置 | 用于配置当前模块信息，可选 |
| <dubbo:registry/> | 注册中心配置 | 用于配置连接注册中心相关信息 |
| <dubbo:monitor/> | 监控中心配置 | 用于配置连接监控中心相关信息，可选 |
| <dubbo:provider/> | 提供方配置 | 当 ProtocolConfig 和 ServiceConfig 某属性没有配置时，采用此缺省值，可选 |
| <dubbo:consumer/> | 消费方配置 | 当 ReferenceConfig 某属性没有配置时，采用此缺省值，可选 |
| <dubbo:method/> | 方法配置 | 用于 ServiceConfig 和 ReferenceConfig 指定方法级的配置信息 |
| <dubbo:argument/> | 参数配置 | 用于指定方法参数配置 |

3、配置覆盖关系

以 timeout 为例，显示了配置的查找顺序，其它 retries, loadbalance, actives 等类似：

* 方法级优先，接口级次之，全局配置再次之。
* 如果级别一样，则消费方优先，提供方次之。

其中，服务提供方配置，通过 URL 经由注册中心传递给消费方。



建议由服务提供方设置超时，因为一个方法需要执行多长时间，服务提供方更清楚，如果一个消费方同时引用多个服务，就不需要关心每个服务的超时设置。

理论上 ReferenceConfig 的非服务标识配置，在 ConsumerConfig，ServiceConfig, ProviderConfig 均可以缺省配置。

#### 属性配置（.properties）

如果公共配置很简单，没有多注册中心，多协议等情况，或者想多个 Spring 容器想共享配置，可以使用 dubbo.properties 作为缺省配置。

Dubbo 将自动加载 classpath 根目录下的 dubbo.properties，可以通过JVM启动参数 -Ddubbo.properties.file=xxx.properties 改变缺省配置位置。

**映射规则**

将 XML 配置的标签名，加属性名，用点分隔，多个属性拆成多行

* 比如：dubbo.application.name=foo等价于<dubbo:application name="foo" />
* 比如：dubbo.registry.address=10.20.153.10:9090等价于<dubbo:registry address="10.20.153.10:9090" />

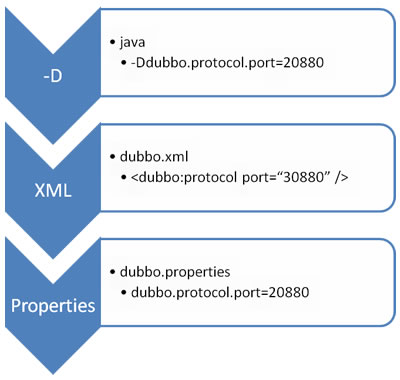
如果 XML 有多行同名标签配置，可用 id 号区分，如果没有 id 号将对所有同名标签生效

* 比如：dubbo.protocol.rmi.port=1234等价于<dubbo:protocol id="rmi" name="rmi" port="1099" /> [[2]](http://dubbo.apache.org/zh-cn/docs/user/configuration/properties.html#fn2)
* 比如：dubbo.registry.china.address=10.20.153.10:9090等价于<dubbo:registry id="china" address="10.20.153.10:9090" />

下面是 dubbo.properties 的一个典型配置：

|  |
| --- |
| dubbo.application.name=foo  dubbo.application.owner=bar  dubbo.registry.address=10.20.153.10:9090 |

**策略覆盖**



JVM 启动 -D 参数优先，这样可以使用户在部署和启动时进行参数重写，比如在启动时需改变协议的端口。

XML 次之，如果在 XML 中有配置，则 dubbo.properties 中的相应配置项无效。

Properties 最后，相当于缺省值，只有 XML 没有配置时，dubbo.properties 的相应配置项才会生效，通常用于共享公共配置，比如应用名。

#### API配置

PI 属性与配置项一对一，各属性含义，请参见：[配置参考手册](http://dubbo.apache.org/zh-cn/docs/user/references/xml/introduction.html)，比如：ApplicationConfig.setName("xxx") 对应 <dubbo:application name="xxx" />

**服务提供者**

|  |
| --- |
| **import** com.alibaba.dubbo.rpc.config.ApplicationConfig;  **import** com.alibaba.dubbo.rpc.config.RegistryConfig;  **import** com.alibaba.dubbo.rpc.config.ProviderConfig;  **import** com.alibaba.dubbo.rpc.config.ServiceConfig;  **import** com.xxx.XxxService;  **import** com.xxx.XxxServiceImpl;    *// 服务实现*  XxxService xxxService = **new** XxxServiceImpl();    *// 当前应用配置*  ApplicationConfig application = **new** ApplicationConfig();  application.setName("xxx");    *// 连接注册中心配置*  RegistryConfig registry = **new** RegistryConfig();  registry.setAddress("10.20.130.230:9090");  registry.setUsername("aaa");  registry.setPassword("bbb");    *// 服务提供者协议配置*  ProtocolConfig protocol = **new** ProtocolConfig();  protocol.setName("dubbo");  protocol.setPort(12345);  protocol.setThreads(200);    *// 注意：ServiceConfig为重对象，内部封装了与注册中心的连接，以及开启服务端口*    *// 服务提供者暴露服务配置*  ServiceConfig<XxxService> service = **new** ServiceConfig<XxxService>(); *// 此实例很重，封装了与注册中心的连接，请自行缓存，否则可能造成内存和连接泄漏*  service.setApplication(application);  service.setRegistry(registry); *// 多个注册中心可以用setRegistries()*  service.setProtocol(protocol); *// 多个协议可以用setProtocols()*  service.setInterface(XxxService.class);  service.setRef(xxxService);  service.setVersion("1.0.0");    *// 暴露及注册服务*  service.export(); |

**服务消费者**

|  |
| --- |
| **import** com.alibaba.dubbo.rpc.config.ApplicationConfig;  **import** com.alibaba.dubbo.rpc.config.RegistryConfig;  **import** com.alibaba.dubbo.rpc.config.ConsumerConfig;  **import** com.alibaba.dubbo.rpc.config.ReferenceConfig;  **import** com.xxx.XxxService;    *// 当前应用配置*  ApplicationConfig application = **new** ApplicationConfig();  application.setName("yyy");    *// 连接注册中心配置*  RegistryConfig registry = **new** RegistryConfig();  registry.setAddress("10.20.130.230:9090");  registry.setUsername("aaa");  registry.setPassword("bbb");    *// 注意：ReferenceConfig为重对象，内部封装了与注册中心的连接，以及与服务提供方的连接*    *// 引用远程服务*  ReferenceConfig<XxxService> reference = **new** ReferenceConfig<XxxService>(); *// 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏*  reference.setApplication(application);  reference.setRegistry(registry); *// 多个注册中心可以用setRegistries()*  reference.setInterface(XxxService.class);  reference.setVersion("1.0.0");    *// 和本地bean一样使用xxxService*  XxxService xxxService = reference.get(); *// 注意：此代理对象内部封装了所有通讯细节，对象较重，请缓存复用* |

**方法级设置**

|  |
| --- |
| ...    *// 方法级配置*  List<MethodConfig> methods = **new** ArrayList<MethodConfig>();  MethodConfig method = **new** MethodConfig();  method.setName("createXxx");  method.setTimeout(10000);  method.setRetries(0);  methods.add(method);    *// 引用远程服务*  ReferenceConfig<XxxService> reference = **new** ReferenceConfig<XxxService>(); *// 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏*  ...  reference.setMethods(methods); *// 设置方法级配置*    ... |

**点对点直连**

|  |
| --- |
| ...    ReferenceConfig<XxxService> reference = **new** ReferenceConfig<XxxService>(); *// 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏*  *// 如果点对点直连，可以用reference.setUrl()指定目标地址，设置url后将绕过注册中心，*  *// 其中，协议对应provider.setProtocol()的值，端口对应provider.setPort()的值，*  *// 路径对应service.setPath()的值，如果未设置path，缺省path为接口名*  reference.setUrl("dubbo://10.20.130.230:20880/com.xxx.XxxService");    ... |

#### 注解配置

需要 2.5.7 及以上版本支持

##### 服务提供方

**Service注解暴露服务**

|  |
| --- |
| **import** com.alibaba.dubbo.config.annotation.Service;    **@Service**(timeout = 5000)  **public** **class** **AnnotateServiceImpl** **implements** **AnnotateService** {  *// ...*  } |

**Javaconfig形式配置公共模块**

|  |
| --- |
| **@Configuration**  **public** **class** **DubboConfiguration** {  **@Bean**  **public** ApplicationConfig **applicationConfig**() {  ApplicationConfig applicationConfig = **new** ApplicationConfig();  applicationConfig.setName("provider-test");  **return** applicationConfig;  }  **@Bean**  **public** RegistryConfig **registryConfig**() {  RegistryConfig registryConfig = **new** RegistryConfig();  registryConfig.setAddress("zookeeper://127.0.0.1:2181");  registryConfig.setClient("curator");  **return** registryConfig;  }  } |

**指定dubbo扫描路径**

|  |
| --- |
| **@SpringBootApplication**  **@DubboComponentScan**(basePackages = "com.alibaba.dubbo.test.service.impl")  **public** **class** **ProviderTestApp** {  *// ...*  } |

##### 服务消费方

**Reference注解引用服务**

|  |
| --- |
| **public** **class** **AnnotationConsumeService** {  **@com**.alibaba.dubbo.config.annotation.Reference  **public** AnnotateService annotateService;    *// ...*  } |

**Javaconfig形式配置公共模块**

|  |
| --- |
| **@Configuration**  **public** **class** **DubboConfiguration** {  **@Bean**  **public** ApplicationConfig **applicationConfig**() {  ApplicationConfig applicationConfig = **new** ApplicationConfig();  applicationConfig.setName("consumer-test");  **return** applicationConfig;  }  **@Bean**  **public** ConsumerConfig **consumerConfig**() {  ConsumerConfig consumerConfig = **new** ConsumerConfig();  consumerConfig.setTimeout(3000);  **return** consumerConfig;  }  **@Bean**  **public** RegistryConfig **registryConfig**() {  RegistryConfig registryConfig = **new** RegistryConfig();  registryConfig.setAddress("zookeeper://127.0.0.1:2181");  registryConfig.setClient("curator");  **return** registryConfig;  }  } |

**指定dubbo扫描路径**

|  |
| --- |
| **@SpringBootApplication**  **@DubboComponentScan**(basePackages = "com.alibaba.dubbo.test.service")  **public** **class** **ConsumerTestApp** {  *// ...*  } |

### 示例

#### 启动时检查

Dubbo 缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，以便上线时，能及早发现问题，默认 check="true"。

可以通过 check="false" 关闭检查，比如，测试时，有些服务不关心，或者出现了循环依赖，必须有一方先启动。

另外，如果你的 Spring 容器是懒加载的，或者通过 API 编程延迟引用服务，请关闭 check，否则服务临时不可用时，会抛出异常，拿到 null 引用，如果 check="false"，总是会返回引用，当服务恢复时，能自动连上。

**通过spring配置文件**

关闭某个服务的启动时检查 (没有提供者时报错)：

<dubbo:reference interface="com.foo.BarService" check="false" />

关闭所有服务的启动时检查 (没有提供者时报错)：

<dubbo:consumer check="false" />

关闭注册中心启动时检查 (注册订阅失败时报错)：

<dubbo:registry check="false" />

**通过dubbo.properties**

dubbo.reference.com.foo.BarService.check=false

dubbo.reference.check=false

dubbo.consumer.check=false

dubbo.registry.check=false

**通过 –D 参数**

java -Ddubbo.reference.com.foo.BarService.check=false

java -Ddubbo.reference.check=false

java -Ddubbo.consumer.check=false

java -Ddubbo.registry.check=false

**配置的含义**

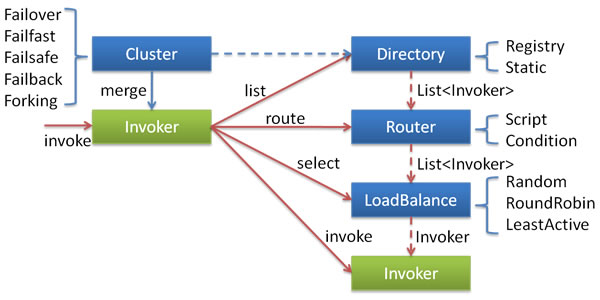
dubbo.reference.check=false，强制改变所有 reference 的 check 值，就算配置中有声明，也会被覆盖。

dubbo.consumer.check=false，是设置 check 的缺省值，如果配置中有显式的声明，如：<dubbo:reference check="true"/>，不会受影响。

dubbo.registry.check=false，前面两个都是指订阅成功，但提供者列表是否为空是否报错，如果注册订阅失败时，也允许启动，需使用此选项，将在后台定时重试。

#### 集群容错

在集群调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。



各节点关系：

* 这里的 Invoker 是 Provider 的一个可调用 Service 的抽象，Invoker 封装了 Provider 地址及Service 接口信息
* Directory 代表多个 Invoker，可以把它看成 List<Invoker> ，但与 List 不同的是，它的值可能是动态变化的，比如注册中心推送变更
* Cluster 将 Directory 中的多个 Invoker 伪装成一个 Invoker，对上层透明，伪装过程包含了容错逻辑，调用失败后，重试另一个
* Router 负责从多个 Invoker 中按路由规则选出子集，比如读写分离，应用隔离等
* LoadBalance 负责从多个 Invoker 中选出具体的一个用于本次调用，选的过程包含了负载均衡算法，调用失败后，需要重选

##### 集群容错模式

可以自行扩展集群容错策略，

Failover Cluster

失败自动切换，当出现失败，重试其它服务器 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/fault-tolerent-strategy.html#fn1)。通常用于读操作，但重试会带来更长延迟。可通过retries="2" 来设置重试次数(不含第一次)。

重试次数配置如下：

<dubbo:service retries="2" />

或

<dubbo:reference retries="2" />

或

<dubbo:reference>

<dubbo:method name="findFoo" retries="2" />

</dubbo:reference>

Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。

Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错 [[2]](http://dubbo.apache.org/zh-cn/docs/user/demos/fault-tolerent-strategy.html#fn2)。通常用于通知所有提供者更新缓存或日志等本地资源信息。

集群模式配置

按照以下示例在服务提供方和消费方配置集群模式

<dubbo:service cluster="failsafe" />

或

<dubbo:reference cluster="failsafe" />

#### 负载均衡

在集群负载均衡时，Dubbo 提供了多种均衡策略，缺省为 random 随机调用。

可以自行扩展负载均衡策略，参见：[负载均衡扩展](http://dubbo.apache.org/zh-cn/docs/dev/impls/load-balance.html)

##### 负载均衡策略

Random LoadBalance

* **随机**，按权重设置随机概率。
* 在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

RoundRobin LoadBalance

* **轮询**，按公约后的权重设置轮询比率。
* 存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

LeastActive LoadBalance

* **最少活跃调用数**，相同活跃数的随机，活跃数指调用前后计数差。
* 使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

ConsistentHash LoadBalance

* **一致性 Hash**，相同参数的请求总是发到同一提供者。
* 当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。
* 算法参见：<http://en.wikipedia.org/wiki/Consistent_hashing>
* 缺省只对第一个参数 Hash，如果要修改，请配置 <dubbo:parameter key="hash.arguments" value="0,1" />
* 缺省用 160 份虚拟节点，如果要修改，请配置 <dubbo:parameter key="hash.nodes" value="320" />

##### 配置

服务端服务级别

<dubbo:service interface="..." loadbalance="roundrobin" />

客户端服务级别

<dubbo:reference interface="..." loadbalance="roundrobin" />

服务端方法级别

<dubbo:service interface="...">

<dubbo:method name="..." loadbalance="roundrobin"/>

</dubbo:service>

客户端方法级别

<dubbo:reference interface="...">

<dubbo:method name="..." loadbalance="roundrobin"/>

</dubbo:reference>

#### 线程模型

如果事件处理的逻辑能迅速完成，并且不会发起新的 IO 请求，比如只是在内存中记个标识，则直接在 IO 线程上处理更快，因为减少了线程池调度。

但如果事件处理逻辑较慢，或者需要发起新的 IO 请求，比如需要查询数据库，则必须派发到线程池，否则 IO 线程阻塞，将导致不能接收其它请求。

如果用 IO 线程处理事件，又在事件处理过程中发起新的 IO 请求，比如在连接事件中发起登录请求，会报“可能引发死锁”异常，但不会真死锁。



因此，需要通过不同的派发策略和不同的线程池配置的组合来应对不同的场景:

<dubbo:protocol name="dubbo" dispatcher="all" threadpool="fixed" threads="100" />

Dispatcher

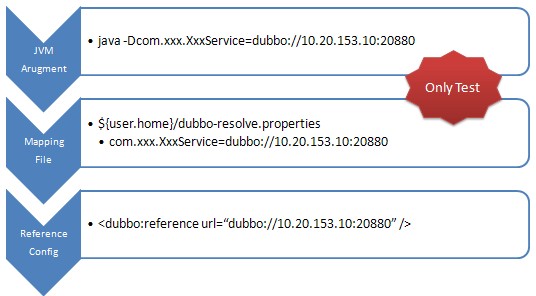
* all 所有消息都派发到线程池，包括请求，响应，连接事件，断开事件，心跳等。
* direct 所有消息都不派发到线程池，全部在 IO 线程上直接执行。
* message 只有请求响应消息派发到线程池，其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
* execution 只请求消息派发到线程池，不含响应，响应和其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
* connection 在 IO 线程上，将连接断开事件放入队列，有序逐个执行，其它消息派发到线程池。

ThreadPool

* fixed 固定大小线程池，启动时建立线程，不关闭，一直持有。(缺省)
* cached 缓存线程池，空闲一分钟自动删除，需要时重建。
* limited 可伸缩线程池，但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题。
* eager 优先创建Worker线程池。在任务数量大于corePoolSize但是小于maximumPoolSize时，优先创建Worker来处理任务。当任务数量大于maximumPoolSize时，将任务放入阻塞队列中。阻塞队列充满时抛出RejectedExecutionException。(相比于cached:cached在任务数量超过maximumPoolSize时直接抛出异常而不是将任务放入阻塞队列)

#### 直连提供者

在开发及测试环境下，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，点对点直连方式，将以服务接口为单位，忽略注册中心的提供者列表，A 接口配置点对点，不影响 B 接口从注册中心获取列表。



通过 XML 配置

如果是线上需求需要点对点，可在 <dubbo:reference> 中配置 url 指向提供者，将绕过注册中心，多个地址用分号隔开，配置如下 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/explicit-target.html#fn1)：

<dubbo:reference id="xxxService" interface="com.alibaba.xxx.XxxService" url="dubbo://localhost:20890" />

通过 -D 参数指定

在 JVM 启动参数中加入-D参数映射服务地址 [[2]](http://dubbo.apache.org/zh-cn/docs/user/demos/explicit-target.html#fn2)，如：

java -Dcom.alibaba.xxx.XxxService=dubbo://localhost:20890

通过文件映射

如果服务比较多，也可以用文件映射，用 -Ddubbo.resolve.file 指定映射文件路径，此配置优先级高于<dubbo:reference> 中的配置 [[3]](http://dubbo.apache.org/zh-cn/docs/user/demos/explicit-target.html#fn3)，如：

java -Ddubbo.resolve.file=xxx.properties

然后在映射文件 xxx.properties 中加入配置，其中 key 为服务名，value 为服务提供者 URL：

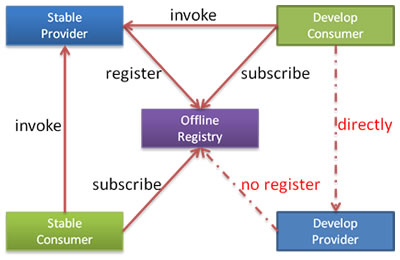
com.alibaba.xxx.XxxService=dubbo://localhost:20890

**注意** 为了避免复杂化线上环境，不要在线上使用这个功能，只应在测试阶段使用。

#### 只订阅

为方便开发测试，经常会在线下共用一个所有服务可用的注册中心，这时，如果一个正在开发中的服务提供者注册，可能会影响消费者不能正常运行。

可以让服务提供者开发方，只订阅服务(开发的服务可能依赖其它服务)，而不注册正在开发的服务，通过直连测试正在开发的服务。



禁用注册配置

<dubbo:registry address="10.20.153.10:9090" register="false" />

或者

<dubbo:registry address="10.20.153.10:9090?register=false" />

#### 只注册

如果有两个镜像环境，两个注册中心，有一个服务只在其中一个注册中心有部署，另一个注册中心还没来得及部署，而两个注册中心的其它应用都需要依赖此服务。这个时候，可以让服务提供者方只注册服务到另一注册中心，而不从另一注册中心订阅服务。

禁用订阅配置

<dubbo:registry id="hzRegistry" address="10.20.153.10:9090" />

<dubbo:registry id="qdRegistry" address="10.20.141.150:9090" subscribe="false" />

或者

<dubbo:registry id="hzRegistry" address="10.20.153.10:9090" />

<dubbo:registry id="qdRegistry" address="10.20.141.150:9090?subscribe=false" />

#### 静态服务

有时候希望人工管理服务提供者的上线和下线，此时需将注册中心标识为非动态管理模式。

<dubbo:registry address="10.20.141.150:9090" dynamic="false" />

或者

<dubbo:registry address="10.20.141.150:9090?dynamic=false" />

服务提供者初次注册时为禁用状态，需人工启用。断线时，将不会被自动删除，需人工禁用。

如果是一个第三方服务提供者，比如 memcached，可以直接向注册中心写入提供者地址信息，消费者正常使用 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/static-service.html#fn1)：

RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();

Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));

registry.register(URL.valueOf("memcached://10.20.153.11/com.foo.BarService?category=providers&dynamic=false&application=foo"));

#### 多协议

Dubbo 允许配置多协议，在不同服务上支持不同协议或者同一服务上同时支持多种协议。

不同服务不同协议

不同服务在性能上适用不同协议进行传输，比如大数据用短连接协议，小数据大并发用长连接协议

**<?**xml version="1.0" encoding="UTF-8"**?>**

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

<dubbo:application name="world" />

<dubbo:registry id="registry" address="10.20.141.150:9090" username="admin" password="hello1234" />

*<!-- 多协议配置 -->*

<dubbo:protocol name="dubbo" port="20880" />

<dubbo:protocol name="rmi" port="1099" />

*<!-- 使用dubbo协议暴露服务 -->*

<dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0" ref="helloService" protocol="dubbo" />

*<!-- 使用rmi协议暴露服务 -->*

<dubbo:service interface="com.alibaba.hello.api.DemoService" version="1.0.0" ref="demoService" protocol="rmi" />

</beans>

多协议暴露服务

需要与 http 客户端互操作

**<?**xml version="1.0" encoding="UTF-8"**?>**

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

<dubbo:application name="world" />

<dubbo:registry id="registry" address="10.20.141.150:9090" username="admin" password="hello1234" />

*<!-- 多协议配置 -->*

<dubbo:protocol name="dubbo" port="20880" />

<dubbo:protocol name="hessian" port="8080" />

*<!-- 使用多个协议暴露服务 -->*

<dubbo:service id="helloService" interface="com.alibaba.hello.api.HelloService" version="1.0.0" protocol="dubbo,hessian" />

</beans>

#### 多注册中心

Dubbo 支持同一服务向多注册中心同时注册，或者不同服务分别注册到不同的注册中心上去，甚至可以同时引用注册在不同注册中心上的同名服务。另外，注册中心是支持自定义扩展的 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/multi-registry.html#fn1)。

多注册中心注册

比如：中文站有些服务来不及在青岛部署，只在杭州部署，而青岛的其它应用需要引用此服务，就可以将服务同时注册到两个注册中心。

**<?**xml version="1.0" encoding="UTF-8"**?>**

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

<dubbo:application name="world" />

*<!-- 多注册中心配置 -->*

<dubbo:registry id="hangzhouRegistry" address="10.20.141.150:9090" />

<dubbo:registry id="qingdaoRegistry" address="10.20.141.151:9010" default="false" />

*<!-- 向多个注册中心注册 -->*

<dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0" ref="helloService" registry="hangzhouRegistry,qingdaoRegistry" />

</beans>

不同服务使用不同注册中心

比如：CRM 有些服务是专门为国际站设计的，有些服务是专门为中文站设计的。

**<?**xml version="1.0" encoding="UTF-8"**?>**

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

<dubbo:application name="world" />

*<!-- 多注册中心配置 -->*

<dubbo:registry id="chinaRegistry" address="10.20.141.150:9090" />

<dubbo:registry id="intlRegistry" address="10.20.154.177:9010" default="false" />

*<!-- 向中文站注册中心注册 -->*

<dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0" ref="helloService" registry="chinaRegistry" />

*<!-- 向国际站注册中心注册 -->*

<dubbo:service interface="com.alibaba.hello.api.DemoService" version="1.0.0" ref="demoService" registry="intlRegistry" />

</beans>

多注册中心引用

比如：CRM 需同时调用中文站和国际站的 PC2 服务，PC2 在中文站和国际站均有部署，接口及版本号都一样，但连的数据库不一样。

**<?**xml version="1.0" encoding="UTF-8"**?>**

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

<dubbo:application name="world" />

*<!-- 多注册中心配置 -->*

<dubbo:registry id="chinaRegistry" address="10.20.141.150:9090" />

<dubbo:registry id="intlRegistry" address="10.20.154.177:9010" default="false" />

*<!-- 引用中文站服务 -->*

<dubbo:reference id="chinaHelloService" interface="com.alibaba.hello.api.HelloService" version="1.0.0" registry="chinaRegistry" />

*<!-- 引用国际站站服务 -->*

<dubbo:reference id="intlHelloService" interface="com.alibaba.hello.api.HelloService" version="1.0.0" registry="intlRegistry" />

</beans>

如果只是测试环境临时需要连接两个不同注册中心，使用竖号分隔多个不同注册中心地址：

**<?**xml version="1.0" encoding="UTF-8"**?>**

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

<dubbo:application name="world" />

*<!-- 多注册中心配置，竖号分隔表示同时连接多个不同注册中心，同一注册中心的多个集群地址用逗号分隔 -->*

<dubbo:registry address="10.20.141.150:9090|10.20.154.177:9010" />

*<!-- 引用服务 -->*

<dubbo:reference id="helloService" interface="com.alibaba.hello.api.HelloService" version="1.0.0" />

</beans>

#### 服务分组

当一个接口有多种实现时，可以用 group 区分。

服务

<dubbo:service group="feedback" interface="com.xxx.IndexService" />

<dubbo:service group="member" interface="com.xxx.IndexService" />

引用

<dubbo:reference id="feedbackIndexService" group="feedback" interface="com.xxx.IndexService" />

<dubbo:reference id="memberIndexService" group="member" interface="com.xxx.IndexService" />

任意组 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/service-group.html#fn1)：

<dubbo:reference id="barService" interface="com.foo.BarService" group="\*" />

#### 多版本

当一个接口实现，出现不兼容升级时，可以用版本号过渡，版本号不同的服务相互间不引用。

可以按照以下的步骤进行版本迁移：

1. 在低压力时间段，先升级一半提供者为新版本
2. 再将所有消费者升级为新版本
3. 然后将剩下的一半提供者升级为新版本

老版本服务提供者配置：

<dubbo:service interface="com.foo.BarService" version="1.0.0" />

新版本服务提供者配置：

<dubbo:service interface="com.foo.BarService" version="2.0.0" />

老版本服务消费者配置：

<dubbo:reference id="barService" interface="com.foo.BarService" version="1.0.0" />

新版本服务消费者配置：

<dubbo:reference id="barService" interface="com.foo.BarService" version="2.0.0" />

如果不需要区分版本，可以按照以下的方式配置 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/multi-versions.html#fn1)：

<dubbo:reference id="barService" interface="com.foo.BarService" version="\*" />

#### 分组聚合

按组合并返回结果 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/group-merger.html#fn1)，比如菜单服务，接口一样，但有多种实现，用group区分，现在消费方需从每种group中调用一次返回结果，合并结果返回，这样就可以实现聚合菜单项。

##### 配置

搜索所有分组

<dubbo:reference interface="com.xxx.MenuService" group="\*" merger="true" />

合并指定分组

<dubbo:reference interface="com.xxx.MenuService" group="aaa,bbb" merger="true" />

指定方法合并结果，其它未指定的方法，将只调用一个 Group

<dubbo:reference interface="com.xxx.MenuService" group="\*">

<dubbo:method name="getMenuItems" merger="true" />

</dubbo:reference>

某个方法不合并结果，其它都合并结果

<dubbo:reference interface="com.xxx.MenuService" group="\*" merger="true">

<dubbo:method name="getMenuItems" merger="false" />

</dubbo:reference>

指定合并策略，缺省根据返回值类型自动匹配，如果同一类型有两个合并器时，需指定合并器的名称 [[2]](http://dubbo.apache.org/zh-cn/docs/user/demos/group-merger.html#fn2)

<dubbo:reference interface="com.xxx.MenuService" group="\*">

<dubbo:method name="getMenuItems" merger="mymerge" />

</dubbo:reference>

指定合并方法，将调用返回结果的指定方法进行合并，合并方法的参数类型必须是返回结果类型本身

<dubbo:reference interface="com.xxx.MenuService" group="\*">

<dubbo:method name="getMenuItems" merger=".addAll" />

</dubbo:reference>

#### 参数验证

##### Maven依赖

|  |
| --- |
| <dependency>  <groupId>javax.validation</groupId>  <artifactId>validation-api</artifactId>  <version>1.0.0.GA</version>  </dependency>  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-validator</artifactId>  <version>4.2.0.Final</version>  </dependency> |

##### 示例

参数标注示例

**import** java.io.Serializable;

**import** java.util.Date;

**import** javax.validation.constraints.Future;

**import** javax.validation.constraints.Max;

**import** javax.validation.constraints.Min;

**import** javax.validation.constraints.NotNull;

**import** javax.validation.constraints.Past;

**import** javax.validation.constraints.Pattern;

**import** javax.validation.constraints.Size;

**public** **class** **ValidationParameter** **implements** **Serializable** {

**private** **static** **final** **long** serialVersionUID = 7158911668568000392L;

**@NotNull** *// 不允许为空*

**@Size**(min = 1, max = 20) *// 长度或大小范围*

**private** String name;

**@NotNull**(groups = ValidationService.Save.class) *// 保存时不允许为空，更新时允许为空 ，表示不更新该字段*

**@Pattern**(regexp = "^\\s\*\\w+(?:\\.{0,1}[\\w-]+)\*@[a-zA-Z0-9]+(?:[-.][a-zA-Z0-9]+)\*\\.[a-zA-Z]+\\s\*$")

**private** String email;

**@Min**(18) *// 最小值*

**@Max**(100) *// 最大值*

**private** **int** age;

**@Past** *// 必须为一个过去的时间*

**private** Date loginDate;

**@Future** *// 必须为一个未来的时间*

**private** Date expiryDate;

**public** String **getName**() {

**return** name;

}

**public** **void** **setName**(String name) {

**this**.name = name;

}

**public** String **getEmail**() {

**return** email;

}

**public** **void** **setEmail**(String email) {

**this**.email = email;

}

**public** **int** **getAge**() {

**return** age;

}

**public** **void** **setAge**(**int** age) {

**this**.age = age;

}

**public** Date **getLoginDate**() {

**return** loginDate;

}

**public** **void** **setLoginDate**(Date loginDate) {

**this**.loginDate = loginDate;

}

**public** Date **getExpiryDate**() {

**return** expiryDate;

}

**public** **void** **setExpiryDate**(Date expiryDate) {

**this**.expiryDate = expiryDate;

}

}

分组验证示例

**public** **interface** **ValidationService** { *// 缺省可按服务接口区分验证场景，如：@NotNull(groups = ValidationService.class)*

**@interface** Save{} *// 与方法同名接口，首字母大写，用于区分验证场景，如：@NotNull(groups = ValidationService.Save.class)，可选*

**void** **save**(ValidationParameter parameter);

**void** **update**(ValidationParameter parameter);

}

关联验证示例

**import** javax.validation.GroupSequence;

**public** **interface** **ValidationService** {

**@GroupSequence**(Update.class) *// 同时验证Update组规则*

**@interface** Save{}

**void** **save**(ValidationParameter parameter);

**@interface** Update{}

**void** **update**(ValidationParameter parameter);

}

参数验证示例

**import** javax.validation.constraints.Min;

**import** javax.validation.constraints.NotNull;

**public** **interface** **ValidationService** {

**void** **save**(@NotNull ValidationParameter parameter); *// 验证参数不为空*

**void** **delete**(@Min(1) **int** id); *// 直接对基本类型参数验证*

}

##### 配置

在客户端验证参数

<dubbo:reference id="validationService" interface="com.alibaba.dubbo.examples.validation.api.ValidationService" validation="true" />

在服务器端验证参数

<dubbo:service interface="com.alibaba.dubbo.examples.validation.api.ValidationService" ref="validationService" validation="true" />

验证异常信息

**import** javax.validation.ConstraintViolationException;

**import** javax.validation.ConstraintViolationException;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** com.alibaba.dubbo.examples.validation.api.ValidationParameter;

**import** com.alibaba.dubbo.examples.validation.api.ValidationService;

**import** com.alibaba.dubbo.rpc.RpcException;

**public** **class** **ValidationConsumer** {

**public** **static** **void** **main**(String[] args) **throws** Exception {

String config = ValidationConsumer.class.getPackage().getName().replace('.', '/') + "/validation-consumer.xml";

ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(config);

context.start();

ValidationService validationService = (ValidationService)context.getBean("validationService");

*// Error*

**try** {

parameter = **new** ValidationParameter();

validationService.save(parameter);

System.out.println("Validation ERROR");

} **catch** (RpcException e) { *// 抛出的是RpcException*

ConstraintViolationException ve = (ConstraintViolationException) e.getCause(); *// 里面嵌了一个ConstraintViolationException*

Set<ConstraintViolation<?>> violations = ve.getConstraintViolations(); *// 可以拿到一个验证错误详细信息的集合*

System.out.println(violations);

}

}

}

#### 结果缓存

结果缓存 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/result-cache.html#fn1)，用于加速热门数据的访问速度，Dubbo 提供声明式缓存，以减少用户加缓存的工作量 [[2]](http://dubbo.apache.org/zh-cn/docs/user/demos/result-cache.html#fn2)。

缓存类型

* lru 基于最近最少使用原则删除多余缓存，保持最热的数据被缓存。
* threadlocal 当前线程缓存，比如一个页面渲染，用到很多 portal，每个 portal 都要去查用户信息，通过线程缓存，可以减少这种多余访问。
* jcache 与 [JSR107](http://jcp.org/en/jsr/detail?id=107%27) 集成，可以桥接各种缓存实现。

缓存类型可扩展，参见：[缓存扩展](http://dubbo.apache.org/books/dubbo-dev-book/impls/cache.html)

配置

<dubbo:reference interface="com.foo.BarService" cache="lru" />

或：

<dubbo:reference interface="com.foo.BarService">

<dubbo:method name="findBar" cache="lru" />

</dubbo:reference>

#### 使用泛化调用

#### 回声测试

回声测试用于检测服务是否可用，回声测试按照正常请求流程执行，能够测试整个调用是否通畅，可用于监控。

所有服务自动实现 EchoService 接口，只需将任意服务引用强制转型为 EchoService，即可使用。

Spring 配置：

<dubbo:reference id="memberService" interface="com.xxx.MemberService" />

代码：

*// 远程服务引用*

MemberService memberService = ctx.getBean("memberService");

EchoService echoService = (EchoService) memberService; *// 强制转型为EchoService*

*// 回声测试可用性*

String status = echoService.$echo("OK");

**assert**(status.equals("OK"));

#### 上下文信息

上下文中存放的是当前调用过程中所需的环境信息。所有配置信息都将转换为 URL 的参数，参见 [schema 配置参考手册](http://dubbo.apache.org/zh-cn/docs/user/references/xml/introduction.html) 中的**对应URL参数**一列。

RpcContext 是一个 ThreadLocal 的临时状态记录器，当接收到 RPC 请求，或发起 RPC 请求时，RpcContext 的状态都会变化。比如：A 调 B，B 再调 C，则 B 机器上，在 B 调 C 之前，RpcContext 记录的是 A 调 B 的信息，在 B 调 C 之后，RpcContext 记录的是 B 调 C 的信息。

服务消费方

*// 远程调用*

xxxService.xxx();

*// 本端是否为消费端，这里会返回true*

**boolean** isConsumerSide = RpcContext.getContext().isConsumerSide();

*// 获取最后一次调用的提供方IP地址*

String serverIP = RpcContext.getContext().getRemoteHost();

*// 获取当前服务配置信息，所有配置信息都将转换为URL的参数*

String application = RpcContext.getContext().getUrl().getParameter("application");

*// 注意：每发起RPC调用，上下文状态会变化*

yyyService.yyy();

服务提供方

**public** **class** **XxxServiceImpl** **implements** **XxxService** {

**public** **void** **xxx**() {

*// 本端是否为提供端，这里会返回true*

**boolean** isProviderSide = RpcContext.getContext().isProviderSide();

*// 获取调用方IP地址*

String clientIP = RpcContext.getContext().getRemoteHost();

*// 获取当前服务配置信息，所有配置信息都将转换为URL的参数*

String application = RpcContext.getContext().getUrl().getParameter("application");

*// 注意：每发起RPC调用，上下文状态会变化*

yyyService.yyy();

*// 此时本端变成消费端，这里会返回false*

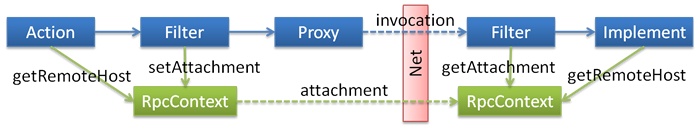
**boolean** isProviderSide = RpcContext.getContext().isProviderSide();

}

}

#### 隐式参数

可以通过 RpcContext 上的 setAttachment 和 getAttachment 在服务消费方和提供方之间进行参数的隐式传递。 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/attachment.html#fn1)



在服务消费方端设置隐式参数

setAttachment 设置的 KV 对，在完成下面一次远程调用会被清空，即多次远程调用要多次设置。

RpcContext.getContext().setAttachment("index", "1"); // 隐式传参，后面的远程调用都会隐式将这些参数发送到服务器端，类似cookie，用于框架集成，不建议常规业务使用

xxxService.xxx(); // 远程调用

// ...

在服务提供方端获取隐式参数

**public** **class** **XxxServiceImpl** **implements** **XxxService** {

**public** **void** **xxx**() {

*// 获取客户端隐式传入的参数，用于框架集成，不建议常规业务使用*

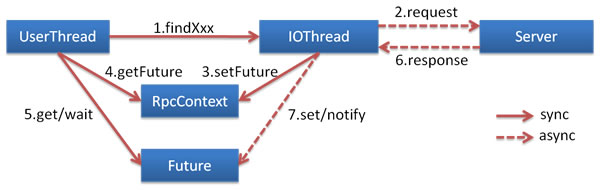
String index = RpcContext.getContext().getAttachment("index");

}

}

#### 异步调用

基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小。 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/async-call.html#fn1)



在 consumer.xml 中配置：

<dubbo:reference id="fooService" interface="com.alibaba.foo.FooService">

<dubbo:method name="findFoo" async="true" />

</dubbo:reference>

<dubbo:reference id="barService" interface="com.alibaba.bar.BarService">

<dubbo:method name="findBar" async="true" />

</dubbo:reference>

调用代码:

*// 此调用会立即返回null*

fooService.findFoo(fooId);

*// 拿到调用的Future引用，当结果返回后，会被通知和设置到此Future*

Future<Foo> fooFuture = RpcContext.getContext().getFuture();

*// 此调用会立即返回null*

barService.findBar(barId);

*// 拿到调用的Future引用，当结果返回后，会被通知和设置到此Future*

Future<Bar> barFuture = RpcContext.getContext().getFuture();

*// 此时findFoo和findBar的请求同时在执行，客户端不需要启动多线程来支持并行，而是借助NIO的非阻塞完成*

*// 如果foo已返回，直接拿到返回值，否则线程wait住，等待foo返回后，线程会被notify唤醒*

Foo foo = fooFuture.get();

*// 同理等待bar返回*

Bar bar = barFuture.get();

*// 如果foo需要5秒返回，bar需要6秒返回，实际只需等6秒，即可获取到foo和bar，进行接下来的处理。*

你也可以设置是否等待消息发出： [[2]](http://dubbo.apache.org/zh-cn/docs/user/demos/async-call.html#fn2)

* sent="true" 等待消息发出，消息发送失败将抛出异常。
* sent="false" 不等待消息发出，将消息放入 IO 队列，即刻返回。

<dubbo:method name="findFoo" async="true" sent="true" />

如果你只是想异步，完全忽略返回值，可以配置 return="false"，以减少 Future 对象的创建和管理成本：

<dubbo:method name="findFoo" async="true" return="false" />

#### 本地调用

本地调用使用了 injvm 协议，是一个伪协议，它不开启端口，不发起远程调用，只在 JVM 内直接关联，但执行 Dubbo 的 Filter 链。

配置

定义 injvm 协议

<dubbo:protocol name="injvm" />

设置默认协议

<dubbo:provider protocol="injvm" />

设置服务协议

<dubbo:service protocol="injvm" />

优先使用 injvm

<dubbo:consumer injvm="true" .../>

<dubbo:provider injvm="true" .../>

或

<dubbo:reference injvm="true" .../>

<dubbo:service injvm="true" .../>

注意：服务暴露与服务引用都需要声明 injvm="true"

自动暴露、引用本地服务

从 2.2.0 开始，每个服务默认都会在本地暴露。在引用服务的时候，默认优先引用本地服务。如果希望引用远程服务可以使用一下配置强制引用远程服务。

<dubbo:reference ... scope="remote" />

#### 参数回调

#### 事件通知

在调用之前、调用之后、出现异常时，会触发 oninvoke、onreturn、onthrow 三个事件，可以配置当事件发生时，通知哪个类的哪个方法 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/events-notify.html#fn1)。

服务提供者与消费者共享服务接口

**interface** **IDemoService** {

**public** Person **get**(**int** id);

}

服务提供者实现

**class** **NormalDemoService** **implements** **IDemoService** {

**public** Person **get**(**int** id) {

**return** **new** Person(id, "charles`son", 4);

}

}

服务提供者配置

<dubbo:application name="rpc-callback-demo" />

<dubbo:registry address="zookeeper://127.0.0.1:2181"/>

<bean id="demoService" class="com.alibaba.dubbo.callback.implicit.NormalDemoService" />

<dubbo:service interface="com.alibaba.dubbo.callback.implicit.IDemoService" ref="demoService" version="1.0.0" group="cn"/>

服务消费者 Callback 接口

**interface** **Notify** {

**public** **void** **onreturn**(Person msg, Integer id);

**public** **void** **onthrow**(Throwable ex, Integer id);

}

服务消费者 Callback 实现

**class** **NotifyImpl** **implements** **Notify** {

**public** Map<Integer, Person> ret = **new** HashMap<Integer, Person>();

**public** Map<Integer, Throwable> errors = **new** HashMap<Integer, Throwable>();

**public** **void** **onreturn**(Person msg, Integer id) {

System.out.println("onreturn:" + msg);

ret.put(id, msg);

}

**public** **void** **onthrow**(Throwable ex, Integer id) {

errors.put(id, ex);

}

}

服务消费者 Callback 配置

<bean id ="demoCallback" class = "com.alibaba.dubbo.callback.implicit.NofifyImpl" />

<dubbo:reference id="demoService" interface="com.alibaba.dubbo.callback.implicit.IDemoService" version="1.0.0" group="cn" >

<dubbo:method name="get" async="true" onreturn = "demoCallback.onreturn" onthrow="demoCallback.onthrow" />

</dubbo:reference>

callback 与 async 功能正交分解，async=true 表示结果是否马上返回，onreturn 表示是否需要回调。

两者叠加存在以下几种组合情况 [[2]](http://dubbo.apache.org/zh-cn/docs/user/demos/events-notify.html#fn2)：

* 异步回调模式：async=true onreturn="xxx"
* 同步回调模式：async=false onreturn="xxx"
* 异步无回调 ：async=true
* 同步无回调 ：async=false

测试代码

IDemoService demoService = (IDemoService) context.getBean("demoService");

NofifyImpl notify = (NofifyImpl) context.getBean("demoCallback");

**int** requestId = 2;

Person ret = demoService.get(requestId);

Assert.assertEquals(**null**, ret);

*//for Test：只是用来说明callback正常被调用，业务具体实现自行决定.*

**for** (**int** i = 0; i < 10; i++) {

**if** (!notify.ret.containsKey(requestId)) {

Thread.sleep(200);

} **else** {

**break**;

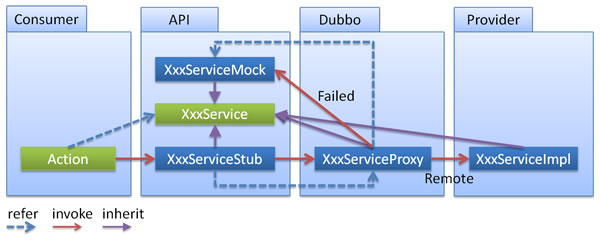
}

}

Assert.assertEquals(requestId, notify.ret.get(requestId).getId());

#### 本地存根

远程服务后，客户端通常只剩下接口，而实现全在服务器端，但提供方有些时候想在客户端也执行部分逻辑，比如：做 ThreadLocal 缓存，提前验证参数，调用失败后伪造容错数据等等，此时就需要在 API 中带上 Stub，客户端生成 Proxy 实例，会把 Proxy 通过构造函数传给 Stub [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/local-stub.html#fn1)，然后把 Stub 暴露给用户，Stub 可以决定要不要去调 Proxy。



在 spring 配置文件中按以下方式配置：

<dubbo:service interface="com.foo.BarService" stub="true" />

或

<dubbo:service interface="com.foo.BarService" stub="com.foo.BarServiceStub" />

提供 Stub 的实现 [[2]](http://dubbo.apache.org/zh-cn/docs/user/demos/local-stub.html#fn2)：

**package** com.foo;

**public** **class** **BarServiceStub** **implements** **BarService** {

**private** **final** BarService barService;

*// 构造函数传入真正的远程代理对象*

**public** (BarService barService) {

**this**.barService = barService;

}

**public** String **sayHello**(String name) {

*// 此代码在客户端执行, 你可以在客户端做ThreadLocal本地缓存，或预先验证参数是否合法，等等*

**try** {

**return** barService.sayHello(name);

} **catch** (Exception e) {

*// 你可以容错，可以做任何AOP拦截事项*

**return** "容错数据";

}

}

}

1. Stub 必须有可传入 Proxy 的构造函数。 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/demos/local-stub.html#fnref1)
2. 在 interface 旁边放一个 Stub 实现，它实现 BarService 接口，并有一个传入远程 BarService 实例的构造函数 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/demos/local-stub.html#fnref2)

#### 本地伪装

本地伪装 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/local-mock.html#fn1) 通常用于服务降级，比如某验权服务，当服务提供方全部挂掉后，客户端不抛出异常，而是通过 Mock 数据返回授权失败。

在 spring 配置文件中按以下方式配置：

<dubbo:reference interface="com.foo.BarService" mock="true" />

或

<dubbo:reference interface="com.foo.BarService" mock="com.foo.BarServiceMock" />

在工程中提供 Mock 实现 [[2]](http://dubbo.apache.org/zh-cn/docs/user/demos/local-mock.html#fn2)：

**package** com.foo;

**public** **class** **BarServiceMock** **implements** **BarService** {

**public** String **sayHello**(String name) {

*// 你可以伪造容错数据，此方法只在出现RpcException时被执行*

**return** "容错数据";

}

}

如果服务的消费方经常需要 try-catch 捕获异常，如：

Offer offer = **null**;

**try** {

offer = offerService.findOffer(offerId);

} **catch** (RpcException e) {

logger.error(e);

}

请考虑改为 Mock 实现，并在 Mock 实现中 return null。如果只是想简单的忽略异常，在 2.0.11 以上版本可用：

<dubbo:reference interface="com.foo.BarService" mock="return null" />

进阶用法

return

使用 return 来返回一个字符串表示的对象，作为 Mock 的返回值。合法的字符串可以是：

* empty: 代表空，基本类型的默认值，或者集合类的空值
* null: null
* true: true
* false: false
* JSON 格式: 反序列化 JSON 所得到的对象

throw

使用 throw 来返回一个 Exception 对象，作为 Mock 的返回值。

当调用出错时，抛出一个默认的 RPCException:

<dubbo:reference interface="com.foo.BarService" mock="throw" />

当调用出错时，抛出指定的 Exception：

<dubbo:reference interface="com.foo.BarService" mock="throw com.foo.MockException" />

force 和 fail

在 2.6.6 以上的版本，可以开始在 Spring XML 配置文件中使用 fail: 和 force:。force: 代表强制使用 Mock 行为，在这种情况下不会走远程调用。fail: 与默认行为一致，只有当远程调用发生错误时才使用 Mock 行为。force: 和 fail: 都支持与 throw 或者 return 组合使用。

强制返回指定值：

<dubbo:reference interface="com.foo.BarService" mock="force:return fake" />

强制抛出指定异常：

<dubbo:reference interface="com.foo.BarService" mock="force:throw com.foo.MockException" />

在方法级别配置 Mock

Mock 可以在方法级别上指定，假定 com.foo.BarService 上有好几个方法，我们可以单独为 sayHello()方法指定 Mock 行为。具体配置如下所示，在本例中，只要 sayHello() 被调用到时，强制返回 "fake":

<dubbo:reference id="demoService" check="false" interface="com.foo.BarService">

<dubbo:parameter key="sayHello.mock" value="force:return fake"/>

</dubbo:reference>

1. Mock 是 Stub 的一个子集，便于服务提供方在客户端执行容错逻辑，因经常需要在出现 RpcException (比如网络失败，超时等)时进行容错，而在出现业务异常(比如登录用户名密码错误)时不需要容错，如果用 Stub，可能就需要捕获并依赖 RpcException 类，而用 Mock 就可以不依赖 RpcException，因为它的约定就是只有出现 RpcException 时才执行。 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/demos/local-mock.html#fnref1)
2. 在 interface 旁放一个 Mock 实现，它实现 BarService 接口，并有一个无参构造函数 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/demos/local-mock.html#fnref2)

#### 延迟暴露

如果你的服务需要预热时间，比如初始化缓存，等待相关资源就位等，可以使用 delay 进行延迟暴露。我们在 Dubbo 2.6.5 版本中对服务延迟暴露逻辑进行了细微的调整，将需要延迟暴露（delay > 0）服务的倒计时动作推迟到了 Spring 初始化完成后进行。你在使用 Dubbo 的过程中，并不会感知到此变化，因此请放心使用。

Dubbo-2.6.5 之前版本

延迟到 Spring 初始化完成后，再暴露服务[[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/delay-publish.html#fn1)

<dubbo:service delay="-1" />

延迟 5 秒暴露服务

<dubbo:service delay="5000" />

Dubbo-2.6.5 及以后版本

所有服务都将在 Spring 初始化完成后进行暴露，如果你不需要延迟暴露服务，无需配置 delay。

延迟 5 秒暴露服务

<dubbo:service delay="5000" />

Spring 2.x 初始化死锁问题

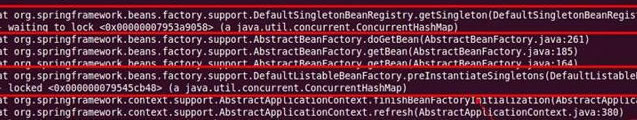
触发条件

在 Spring 解析到 <dubbo:service /> 时，就已经向外暴露了服务，而 Spring 还在接着初始化其它 Bean。如果这时有请求进来，并且服务的实现类里有调用 applicationContext.getBean() 的用法。

1. 请求线程的 applicationContext.getBean() 调用，先同步 singletonObjects 判断 Bean 是否存在，不存在就同步 beanDefinitionMap 进行初始化，并再次同步 singletonObjects 写入 Bean 实例缓存。



1. 而 Spring 初始化线程，因不需要判断 Bean 的存在，直接同步 beanDefinitionMap 进行初始化，并同步 singletonObjects 写入 Bean 实例缓存。



这样就导致 getBean 线程，先锁 singletonObjects，再锁 beanDefinitionMap，再次锁 singletonObjects。  
而 Spring 初始化线程，先锁 beanDefinitionMap，再锁 singletonObjects。反向锁导致线程死锁，不能提供服务，启动不了。

规避办法

1. 强烈建议不要在服务的实现类中有 applicationContext.getBean() 的调用，全部采用 IoC 注入的方式使用 Spring的Bean。
2. 如果实在要调 getBean()，可以将 Dubbo 的配置放在 Spring 的最后加载。
3. 如果不想依赖配置顺序，可以使用 <dubbo:provider delay=”-1” />，使 Dubbo 在 Spring 容器初始化完后，再暴露服务。
4. 如果大量使用 getBean()，相当于已经把 Spring 退化为工厂模式在用，可以将 Dubbo 的服务隔离单独的 Spring 容器。

#### 并发控制

配置样例

样例 1

限制 com.foo.BarService 的每个方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个：

<dubbo:service interface="com.foo.BarService" executes="10" />

样例 2

限制 com.foo.BarService 的 sayHello 方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个：

<dubbo:service interface="com.foo.BarService">

<dubbo:method name="sayHello" executes="10" />

</dubbo:service>

样例 3

限制 com.foo.BarService 的每个方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个：

<dubbo:service interface="com.foo.BarService" actives="10" />

或

<dubbo:reference interface="com.foo.BarService" actives="10" />

样例 4

限制 com.foo.BarService 的 sayHello 方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个：

<dubbo:service interface="com.foo.BarService">

<dubbo:method name="sayHello" actives="10" />

</dubbo:service>

或

<dubbo:reference interface="com.foo.BarService">

<dubbo:method name="sayHello" actives="10" />

</dubbo:service>

如果 <dubbo:service> 和 <dubbo:reference> 都配了actives，<dubbo:reference> 优先，参见：[配置的覆盖策略](http://dubbo.apache.org/zh-cn/docs/user/configuration/xml.md#%E9%85%8D%E7%BD%AE%E8%A6%86%E7%9B%96%E5%85%B3%E7%B3%BB)。

Load Balance 均衡

配置服务的客户端的 loadbalance 属性为 leastactive，此 Loadbalance 会调用并发数最小的 Provider（Consumer端并发数）。

<dubbo:reference interface="com.foo.BarService" loadbalance="leastactive" />

或

<dubbo:service interface="com.foo.BarService" loadbalance="leastactive" />

#### 连接控制

服务端连接控制

限制服务器端接受的连接不能超过 10 个 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/config-connections.html#fn1)：

<dubbo:provider protocol="dubbo" accepts="10" />

或

<dubbo:protocol name="dubbo" accepts="10" />

客户端连接控制

限制客户端服务使用连接不能超过 10 个 [[2]](http://dubbo.apache.org/zh-cn/docs/user/demos/config-connections.html#fn2)：

<dubbo:reference interface="com.foo.BarService" connections="10" />

或

<dubbo:service interface="com.foo.BarService" connections="10" />

如果 <dubbo:service> 和 <dubbo:reference> 都配了 connections，<dubbo:reference> 优先，参见：[配置的覆盖策略](http://dubbo.apache.org/zh-cn/docs/user/configuration/xml.html)

1. 因为连接在 Server上，所以配置在 Provider 上 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/demos/config-connections.html#fnref1)
2. 如果是长连接，比如 Dubbo 协议，connections 表示该服务对每个提供者建立的长连接数 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/demos/config-connections.html#fnref2)

#### 延迟连接

延迟连接用于减少长连接数。当有调用发起时，再创建长连接。[[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/lazy-connect.html#fn1)

<dubbo:protocol name="dubbo" lazy="true" />

#### 粘滞连接

粘滞连接用于有状态服务，尽可能让客户端总是向同一提供者发起调用，除非该提供者挂了，再连另一台。

粘滞连接将自动开启[延迟连接](http://dubbo.apache.org/zh-cn/docs/user/demos/lazy-connect.html)，以减少长连接数。

<dubbo:protocol name="dubbo" sticky="true" />

#### 令牌验证

通过令牌验证在注册中心控制权限，以决定要不要下发令牌给消费者，可以防止消费者绕过注册中心访问提供者，另外通过注册中心可灵活改变授权方式，而不需修改或升级提供者



可以全局设置开启令牌验证：

*<!--随机token令牌，使用UUID生成-->*

<dubbo:provider interface="com.foo.BarService" token="true" />

或

*<!--固定token令牌，相当于密码-->*

<dubbo:provider interface="com.foo.BarService" token="123456" />

也可在服务级别设置：

*<!--随机token令牌，使用UUID生成-->*

<dubbo:service interface="com.foo.BarService" token="true" />

或

*<!--固定token令牌，相当于密码-->*

<dubbo:service interface="com.foo.BarService" token="123456" />

还可在协议级别设置：

*<!--随机token令牌，使用UUID生成-->*

<dubbo:protocol name="dubbo" token="true" />

或

*<!--固定token令牌，相当于密码-->*

<dubbo:protocol name="dubbo" token="123456" />

#### 路由规则

路由规则 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/routing-rule.html#fn1) 决定一次 dubbo 服务调用的目标服务器，分为条件路由规则和脚本路由规则，并且支持可扩展[[2]](http://dubbo.apache.org/zh-cn/docs/user/demos/routing-rule.html#fn2)。

写入路由规则

向注册中心写入路由规则的操作通常由监控中心或治理中心的页面完成

RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();

Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));

registry.register(URL.valueOf("condition://0.0.0.0/com.foo.BarService?category=routers&dynamic=false&rule=" + URL.encode("host = 10.20.153.10 => host = 10.20.153.11")));

其中：

* condition:// 表示路由规则的类型，支持条件路由规则和脚本路由规则，可扩展，**必填**。
* 0.0.0.0 表示对所有 IP 地址生效，如果只想对某个 IP 的生效，请填入具体 IP，**必填**。
* com.foo.BarService 表示只对指定服务生效，**必填**。
* group=foo 对指定服务的指定group生效，不填表示对未配置group的指定服务生效
* version=1.0对指定服务的指定version生效，不填表示对未配置version的指定服务生效
* category=routers 表示该数据为动态配置类型，**必填**。
* dynamic=false 表示该数据为持久数据，当注册方退出时，数据依然保存在注册中心，**必填**。
* enabled=true 覆盖规则是否生效，可不填，缺省生效。
* force=false 当路由结果为空时，是否强制执行，如果不强制执行，路由结果为空的路由规则将自动失效，可不填，缺省为 false。
* runtime=false 是否在每次调用时执行路由规则，否则只在提供者地址列表变更时预先执行并缓存结果，调用时直接从缓存中获取路由结果。如果用了参数路由，必须设为 true，需要注意设置会影响调用的性能，可不填，缺省为 false。
* priority=1 路由规则的优先级，用于排序，优先级越大越靠前执行，可不填，缺省为 0。
* rule=URL.encode("host = 10.20.153.10 => host = 10.20.153.11") 表示路由规则的内容，**必填**。

条件路由规则

基于条件表达式的路由规则，如：host = 10.20.153.10 => host = 10.20.153.11

规则：

* => 之前的为消费者匹配条件，所有参数和消费者的 URL 进行对比，当消费者满足匹配条件时，对该消费者执行后面的过滤规则。
* => 之后为提供者地址列表的过滤条件，所有参数和提供者的 URL 进行对比，消费者最终只拿到过滤后的地址列表。
* 如果匹配条件为空，表示对所有消费方应用，如：=> host != 10.20.153.11
* 如果过滤条件为空，表示禁止访问，如：host = 10.20.153.10 =>

表达式：

参数支持：

* 服务调用信息，如：method, argument 等，暂不支持参数路由
* URL 本身的字段，如：protocol, host, port 等
* 以及 URL 上的所有参数，如：application, organization 等

条件支持：

* 等号 = 表示"匹配"，如：host = 10.20.153.10
* 不等号 != 表示"不匹配"，如：host != 10.20.153.10

值支持：

* 以逗号 , 分隔多个值，如：host != 10.20.153.10,10.20.153.11
* 以星号 \* 结尾，表示通配，如：host != 10.20.\*
* 以美元符 $ 开头，表示引用消费者参数，如：host = $host

示例：

1. 排除预发布机：
2. => host != 172.22.3.91
3. 白名单 [[3]](http://dubbo.apache.org/zh-cn/docs/user/demos/routing-rule.html#fn3)：
4. host != 10.20.153.10,10.20.153.11 =>
5. 黑名单：
6. host = 10.20.153.10,10.20.153.11 =>
7. 服务寄宿在应用上，只暴露一部分的机器，防止整个集群挂掉：
8. => host = 172.22.3.1\*,172.22.3.2\*
9. 为重要应用提供额外的机器：
10. application != kylin => host != 172.22.3.95,172.22.3.96
11. 读写分离：
12. method = find\*,list\*,get\*,is\* => host = 172.22.3.94,172.22.3.95,172.22.3.96
13. method != find\*,list\*,get\*,is\* => host = 172.22.3.97,172.22.3.98
14. 前后台分离：
15. application = bops => host = 172.22.3.91,172.22.3.92,172.22.3.93
16. application != bops => host = 172.22.3.94,172.22.3.95,172.22.3.96
17. 隔离不同机房网段：
18. host != 172.22.3.\* => host != 172.22.3.\*
19. 提供者与消费者部署在同集群内，本机只访问本机的服务：
20. => host = $host

脚本路由规则

脚本路由规则 [[4]](http://dubbo.apache.org/zh-cn/docs/user/demos/routing-rule.html#fn4) 支持 JDK 脚本引擎的所有脚本，比如：javascript, jruby, groovy 等，通过 type=javascript参数设置脚本类型，缺省为 javascript。

"script://0.0.0.0/com.foo.BarService?category=routers&dynamic=false&rule=" + URL.encode("（function route(invokers) { ... } (invokers)）")

基于脚本引擎的路由规则，如：

（**function** **route**(invokers) {

**var** result = **new** java.util.ArrayList(invokers.size());

**for** (i = 0; i < invokers.size(); i ++) {

**if** ("10.20.153.10".equals(invokers.get(i).getUrl().getHost())) {

result.add(invokers.get(i));

}

}

**return** result;

} (invokers)）; *// 表示立即执行方法*

标签路由规则

标签路由规则 [[5]](http://dubbo.apache.org/zh-cn/docs/user/demos/routing-rule.html#fn5) ，当应用选择装配标签路由(TagRouter)之后，一次 dubbo 调用能够根据请求携带的 tag 标签智能地选择对应 tag 的服务提供者进行调用。

服务提供者

1. 给应用装配标签路由器：

**@Bean**

**public** ApplicationConfig **applicationConfig**() {

ApplicationConfig applicationConfig = **new** ApplicationConfig();

applicationConfig.setName("provider-book");

applicationConfig.setQosEnable(**false**);

*// instruct tag router*

Map<String,String> parameters = **new** HashMap<>();

parameters.put(Constants.ROUTER\_KEY, "tag");

applicationConfig.setParameters(parameters);

**return** applicationConfig;

}

1. 给应用设置具体的标签：

**@Bean**

**public** ProviderConfig **providerConfig**(){

ProviderConfig providerConfig = **new** ProviderConfig();

providerConfig.setTag("red");

**return** providerConfig;

}

应用未装配 tag 属性或服务提供者未设置 tag 属性，都将被认为是默认的应用，这些默认应用将会在调用无法匹配 provider 时当作降级方案。

服务消费者

RpcContext.getContext().setAttachment(Constants.REQUEST\_TAG\_KEY,"red");

请求标签的作用域为每一次 invocation，使用 attachment 来传递请求标签，注意保存在 attachment 中的值将会在一次完整的远程调用中持续传递，得益于这样的特性，我们只需要在起始调用时，通过一行代码的设置，达到标签的持续传递。

目前仅仅支持 hardcoding 的方式设置 requestTag。注意到 RpcContext 是线程绑定的，优雅的使用 TagRouter 特性，建议通过 servlet 过滤器(在 web 环境下)，或者定制的 SPI 过滤器设置 requestTag。

规则描述

1. request.tag=red 时优先选择 tag=red 的 provider。若集群中不存在与请求标记对应的服务，可以降级请求 tag=null 的 provider，即默认 provider。
2. request.tag=null 时，只会匹配 tag=null 的 provider。即使集群中存在可用的服务，若 tag 不匹配就无法调用，这与规则1不同，携带标签的请求可以降级访问到无标签的服务，但不携带标签/携带其他种类标签的请求永远无法访问到其他标签的服务。
3. 2.2.0 以上版本支持 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/demos/routing-rule.html#fnref1)
4. 路由规则扩展点：[路由扩展](http://dubbo.apache.org/books/dubbo-dev-book/impls/router.html) [↩︎](http://dubbo.apache.org/zh-cn/docs/user/demos/routing-rule.html#fnref2)
5. 注意：一个服务只能有一条白名单规则，否则两条规则交叉，就都被筛选掉了 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/demos/routing-rule.html#fnref3)
6. 注意：脚本没有沙箱约束，可执行任意代码，存在后门风险 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/demos/routing-rule.html#fnref4)
7. 该特性在 2.7.0 以上版本支持 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/demos/routing-rule.html#fnref5)

#### 配置规则

向注册中心写入动态配置覆盖规则 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/config-rule.html#fn1)。该功能通常由监控中心或治理中心的页面完成。

RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();

Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));

registry.register(URL.valueOf("override://0.0.0.0/com.foo.BarService?category=configurators&dynamic=false&application=foo&timeout=1000"));

其中：

* override:// 表示数据采用覆盖方式，支持 override 和 absent，可扩展，**必填**。
* 0.0.0.0 表示对所有 IP 地址生效，如果只想覆盖某个 IP 的数据，请填入具体 IP，**必填**。
* com.foo.BarService 表示只对指定服务生效，**必填**。
* category=configurators 表示该数据为动态配置类型，**必填**。
* dynamic=false 表示该数据为持久数据，当注册方退出时，数据依然保存在注册中心，**必填**。
* enabled=true 覆盖规则是否生效，可不填，缺省生效。
* application=foo 表示只对指定应用生效，可不填，表示对所有应用生效。
* timeout=1000 表示将满足以上条件的 timeout 参数的值覆盖为 1000。如果想覆盖其它参数，直接加在 override 的 URL 参数上。

示例：

1. 禁用提供者：(通常用于临时踢除某台提供者机器，相似的，禁止消费者访问请使用路由规则)
2. override://10.20.153.10/com.foo.BarService?category=configurators&dynamic=false&disbaled=true
3. 调整权重：(通常用于容量评估，缺省权重为 100)
4. override://10.20.153.10/com.foo.BarService?category=configurators&dynamic=false&weight=200
5. 调整负载均衡策略：(缺省负载均衡策略为 random)
6. override://10.20.153.10/com.foo.BarService?category=configurators&dynamic=false&loadbalance=leastactive
7. 服务降级：(通常用于临时屏蔽某个出错的非关键服务)
8. override://0.0.0.0/com.foo.BarService?category=configurators&dynamic=false&application=foo&mock=force:return+null

#### 服务降级

可以通过服务降级功能 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/service-downgrade.html#fn1) 临时屏蔽某个出错的非关键服务，并定义降级后的返回策略。

向注册中心写入动态配置覆盖规则：

RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();

Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));

registry.register(URL.valueOf("override://0.0.0.0/com.foo.BarService?category=configurators&dynamic=false&application=foo&mock=force:return+null"));

其中：

* mock=force:return+null 表示消费方对该服务的方法调用都直接返回 null 值，不发起远程调用。用来屏蔽不重要服务不可用时对调用方的影响。
* 还可以改为 mock=fail:return+null 表示消费方对该服务的方法调用在失败后，再返回 null 值，不抛异常。用来容忍不重要服务不稳定时对调用方的影响。

#### 优雅停机

Dubbo 是通过 JDK 的 ShutdownHook 来完成优雅停机的，所以如果用户使用 kill -9 PID 等强制关闭指令，是不会执行优雅停机的，只有通过 kill PID 时，才会执行。

原理

服务提供方

* 停止时，先标记为不接收新请求，新请求过来时直接报错，让客户端重试其它机器。
* 然后，检测线程池中的线程是否正在运行，如果有，等待所有线程执行完成，除非超时，则强制关闭。

服务消费方

* 停止时，不再发起新的调用请求，所有新的调用在客户端即报错。
* 然后，检测有没有请求的响应还没有返回，等待响应返回，除非超时，则强制关闭。

设置方式

设置优雅停机超时时间，缺省超时时间是 10 秒，如果超时则强制关闭。

# dubbo.properties

dubbo.service.shutdown.wait=15000

如果 ShutdownHook 不能生效，可以自行调用，**使用tomcat等容器部署的場景，建议通过扩展ContextListener等自行调用以下代码实现优雅停机**：

ProtocolConfig.destroyAll();

#### 主机绑定

查找顺序

缺省主机 IP 查找顺序：

* 通过 LocalHost.getLocalHost() 获取本机地址。
* 如果是 127.\* 等 loopback 地址，则扫描各网卡，获取网卡 IP。

主机配置

注册的地址如果获取不正确，比如需要注册公网地址，可以：

1. 可以在 /etc/hosts 中加入：机器名 公网 IP，比如：
2. test1 205.182.23.201
3. 在 dubbo.xml 中加入主机地址的配置：
4. <dubbo:protocol host="205.182.23.201">
5. 或在 dubbo.properties 中加入主机地址的配置：
6. dubbo.protocol.host=205.182.23.201

端口配置

缺省主机端口与协议相关：

| **协议** | **端口** |
| --- | --- |
| dubbo | 20880 |
| rmi | 1099 |
| http | 80 |
| hessian | 80 |
| webservice | 80 |
| memcached | 11211 |
| redis | 6379 |

可以按照下面的方式配置端口：

1. 在 dubbo.xml 中加入主机地址的配置：
2. <dubbo:protocol name="dubbo" port="20880">
3. 或在 dubbo.properties 中加入主机地址的配置：

dubbo.protocol.dubbo.port=20880

#### 日志适配

自 2.2.1 开始，dubbo 开始内置 log4j、slf4j、jcl、jdk 这些日志框架的适配[1]，也可以通过以下方式显示配置日志输出策略：

1. 命令行
2. java -Ddubbo.application.logger=log4j
3. 在 dubbo.properties 中指定
4. dubbo.application.logger=log4j
5. 在 dubbo.xml 中配置

<dubbo:application logger="log4j" />

#### 访问日志

如果你想记录每一次请求信息，可开启访问日志，类似于apache的访问日志。**注意**：此日志量比较大，请注意磁盘容量。

将访问日志输出到当前应用的log4j日志：

<dubbo:protocol accesslog="true" />

将访问日志输出到指定文件：

<dubbo:protocol accesslog="http://10.20.160.198/wiki/display/dubbo/foo/bar.log" />

#### 服务容器

服务容器是一个 standalone 的启动程序，因为后台服务不需要 Tomcat 或 JBoss 等 Web 容器的功能，如果硬要用 Web 容器去加载服务提供方，增加复杂性，也浪费资源。

服务容器只是一个简单的 Main 方法，并加载一个简单的 Spring 容器，用于暴露服务。

服务容器的加载内容可以扩展，内置了 spring, jetty, log4j 等加载，可通过[容器扩展点](http://dubbo.apache.org/books/dubbo-dev-book/impls/container.html)进行扩展。配置配在 java 命令的 -D 参数或者 dubbo.properties 中。

容器类型

Spring Container

* 自动加载 META-INF/spring 目录下的所有 Spring 配置。
* 配置 spring 配置加载位置：
* dubbo.spring.config=classpath\*:META-INF/spring/\*.xml

Jetty Container

* 启动一个内嵌 Jetty，用于汇报状态。
* 配置：
  + dubbo.jetty.port=8080：配置 jetty 启动端口
  + dubbo.jetty.directory=/foo/bar：配置可通过 jetty 直接访问的目录，用于存放静态文件
  + dubbo.jetty.page=log,status,system：配置显示的页面，缺省加载所有页面

Log4j Container

* 自动配置 log4j 的配置，在多进程启动时，自动给日志文件按进程分目录。
* 配置：
  + dubbo.log4j.file=/foo/bar.log：配置日志文件路径
  + dubbo.log4j.level=WARN：配置日志级别
  + dubbo.log4j.subdirectory=20880：配置日志子目录，用于多进程启动，避免冲突

容器启动

缺省只加载 spring

java com.alibaba.dubbo.container.Main

通过 main 函数参数传入要加载的容器

java com.alibaba.dubbo.container.Main spring jetty log4j

通过 JVM 启动参数传入要加载的容器

java com.alibaba.dubbo.container.Main -Ddubbo.container=spring,jetty,log4j

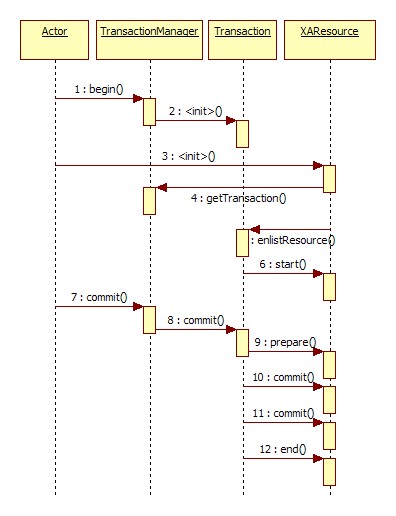
通过 classpath 下的 dubbo.properties 配置传入要加载的容器

dubbo.container=spring,jetty,log4j

#### 分布式事物

分布式事务基于 JTA/XA 规范实现 [[1]](http://dubbo.apache.org/zh-cn/docs/user/demos/distributed-transaction.html#fn1)。

两阶段提交：



1. **本功能暂未实现**[**↩︎**](http://dubbo.apache.org/zh-cn/docs/user/demos/distributed-transaction.html#fnref1)

#### 在Dubbo中使用高效的java序列化（Kryo与FST）

### Schema配置参考手册

#### 介绍

这里以 XML Config [[1]](http://dubbo.apache.org/zh-cn/docs/user/references/xml/introduction.html#fn1) 为准，列举所有配置项 [[2]](http://dubbo.apache.org/zh-cn/docs/user/references/xml/introduction.html#fn2)。其它配置方式，请参见相应转换关系：[属性配置](http://dubbo.apache.org/zh-cn/docs/user/configuration/properties.html)，[注解配置](http://dubbo.apache.org/zh-cn/docs/user/configuration/annotation.html)，[API 配置](http://dubbo.apache.org/zh-cn/docs/user/configuration/api.html)。

所有配置项分为三大类，参见下表中的"作用" 一列。

* 服务发现：表示该配置项用于服务的注册与发现，目的是让消费方找到提供方。
* 服务治理：表示该配置项用于治理服务间的关系，或为开发测试提供便利条件。
* 性能调优：表示该配置项用于调优性能，不同的选项对性能会产生影响。
* 所有配置最终都将转换为 URL [[3]](http://dubbo.apache.org/zh-cn/docs/user/references/xml/introduction.html#fn3) 表示，并由服务提供方生成，经注册中心传递给消费方，各属性对应 URL 的参数，参见配置项一览表中的 "对应URL参数" 列。

1. XML Schema: <http://dubbo.apache.org/schema/dubbo/dubbo.xsd> [↩︎](http://dubbo.apache.org/zh-cn/docs/user/references/xml/introduction.html#fnref1)
2. 注意：只有 group，interface，version 是服务的匹配条件，三者决定是不是同一个服务，其它配置项均为调优和治理参数。 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/references/xml/introduction.html#fnref2)
3. URL 格式：protocol://username:password@host:port/path?key=value&key=value [↩︎](http://dubbo.apache.org/zh-cn/docs/user/references/xml/introduction.html#fnref3)

#### Dubbo:service

服务提供者暴露服务配置。对应的配置类：com.alibaba.dubbo.config.ServiceConfig

| **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| interface |  | class | **必填** |  | 服务发现 | 服务接口名 | 1.0.0以上版本 |
| ref |  | object | **必填** |  | 服务发现 | 服务对象实现引用 | 1.0.0以上版本 |
| version | version | string | 可选 | 0.0.0 | 服务发现 | 服务版本，建议使用两位数字版本，如：1.0，通常在接口不兼容时版本号才需要升级 | 1.0.0以上版本 |
| group | group | string | 可选 |  | 服务发现 | 服务分组，当一个接口有多个实现，可以用分组区分 | 1.0.7以上版本 |
| path | <path> | string | 可选 | 缺省为接口名 | 服务发现 | 服务路径 (注意：1.0不支持自定义路径，总是使用接口名，如果有1.0调2.0，配置服务路径可能不兼容) | 1.0.12以上版本 |
| delay | delay | int | 可选 | 0 | 性能调优 | 延迟注册服务时间(毫秒) ，设为-1时，表示延迟到Spring容器初始化完成时暴露服务 | 1.0.14以上版本 |
| timeout | timeout | int | 可选 | 1000 | 性能调优 | 远程服务调用超时时间(毫秒) | 2.0.0以上版本 |
| retries | retries | int | 可选 | 2 | 性能调优 | 远程服务调用重试次数，不包括第一次调用，不需要重试请设为0 | 2.0.0以上版本 |
| connections | connections | int | 可选 | 100 | 性能调优 | 对每个提供者的最大连接数，rmi、http、hessian等短连接协议表示限制连接数，dubbo等长连接协表示建立的长连接个数 | 2.0.0以上版本 |
| loadbalance | loadbalance | string | 可选 | random | 性能调优 | 负载均衡策略，可选值：random,roundrobin,leastactive，分别表示：随机，轮询，最少活跃调用 | 2.0.0以上版本 |
| async | async | boolean | 可选 | false | 性能调优 | 是否缺省异步执行，不可靠异步，只是忽略返回值，不阻塞执行线程 | 2.0.0以上版本 |
| stub | stub | class/boolean | 可选 | false | 服务治理 | 设为true，表示使用缺省代理类名，即：接口名 + Local后缀，服务接口客户端本地代理类名，用于在客户端执行本地逻辑，如本地缓存等，该本地代理类的构造函数必须允许传入远程代理对象，构造函数如：public XxxServiceLocal(XxxService xxxService) | 2.0.0以上版本 |
| mock | mock | class/boolean | 可选 | false | 服务治理 | 设为true，表示使用缺省Mock类名，即：接口名 + Mock后缀，服务接口调用失败Mock实现类，该Mock类必须有一个无参构造函数，与Local的区别在于，Local总是被执行，而Mock只在出现非业务异常(比如超时，网络异常等)时执行，Local在远程调用之前执行，Mock在远程调用后执行。 | 2.0.0以上版本 |
| token | token | string/boolean | 可选 | false | 服务治理 | 令牌验证，为空表示不开启，如果为true，表示随机生成动态令牌，否则使用静态令牌，令牌的作用是防止消费者绕过注册中心直接访问，保证注册中心的授权功能有效，如果使用点对点调用，需关闭令牌功能 | 2.0.0以上版本 |
| registry |  | string | 可选 | 缺省向所有registry注册 | 配置关联 | 向指定注册中心注册，在多个注册中心时使用，值为<dubbo:registry>的id属性，多个注册中心ID用逗号分隔，如果不想将该服务注册到任何registry，可将值设为N/A | 2.0.0以上版本 |
| provider |  | string | 可选 | 缺使用第一个provider配置 | 配置关联 | 指定provider，值为<dubbo:provider>的id属性 | 2.0.0以上版本 |
| deprecated | deprecated | boolean | 可选 | false | 服务治理 | 服务是否过时，如果设为true，消费方引用时将打印服务过时警告error日志 | 2.0.5以上版本 |
| dynamic | dynamic | boolean | 可选 | true | 服务治理 | 服务是否动态注册，如果设为false，注册后将显示后disable状态，需人工启用，并且服务提供者停止时，也不会自动取消册，需人工禁用。 | 2.0.5以上版本 |
| accesslog | accesslog | string/boolean | 可选 | false | 服务治理 | 设为true，将向logger中输出访问日志，也可填写访问日志文件路径，直接把访问日志输出到指定文件 | 2.0.5以上版本 |
| owner | owner | string | 可选 |  | 服务治理 | 服务负责人，用于服务治理，请填写负责人公司邮箱前缀 | 2.0.5以上版本 |
| document | document | string | 可选 |  | 服务治理 | 服务文档URL | 2.0.5以上版本 |
| weight | weight | int | 可选 |  | 性能调优 | 服务权重 | 2.0.5以上版本 |
| executes | executes | int | 可选 | 0 | 性能调优 | 服务提供者每服务每方法最大可并行执行请求数 | 2.0.5以上版本 |
| proxy | proxy | string | 可选 | javassist | 性能调优 | 生成动态代理方式，可选：jdk/javassist | 2.0.5以上版本 |
| cluster | cluster | string | 可选 | failover | 性能调优 | 集群方式，可选：failover/failfast/failsafe/failback/forking | 2.0.5以上版本 |
| filter | service.filter | string | 可选 | default | 性能调优 | 服务提供方远程调用过程拦截器名称，多个名称用逗号分隔 | 2.0.5以上版本 |
| listener | exporter.listener | string | 可选 | default | 性能调优 | 服务提供方导出服务监听器名称，多个名称用逗号分隔 |  |
| protocol |  | string | 可选 |  | 配置关联 | 使用指定的协议暴露服务，在多协议时使用，值为<dubbo:protocol>的id属性，多个协议ID用逗号分隔 | 2.0.5以上版本 |
| layer | layer | string | 可选 |  | 服务治理 | 服务提供者所在的分层。如：biz、dao、intl:web、china:acton。 | 2.0.7以上版本 |
| register | register | boolean | 可选 | true | 服务治理 | 该协议的服务是否注册到注册中心 | 2.0.8以上版本 |

#### Dubbo:reference

服务消费者引用服务配置。对应的配置类： com.alibaba.dubbo.config.ReferenceConfig

#### Dubbo:protocol

服务提供者协议配置。对应的配置类： com.alibaba.dubbo.config.ProtocolConfig。同时，如果需要支持多协议，可以声明多个 <dubbo:protocol> 标签，并在 <dubbo:service> 中通过 protocol 属性指定使用的协议。

#### Dubbo:registry

注册中心配置。对应的配置类： com.alibaba.dubbo.config.RegistryConfig。同时如果有多个不同的注册中心，可以声明多个 <dubbo:registry> 标签，并在 <dubbo:service> 或 <dubbo:reference> 的registry 属性指定使用的注册中心。

#### Dubbo:monitor

监控中心配置。对应的配置类： com.alibaba.dubbo.config.MonitorConfig

| **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| protocol | protocol | string | 可选 | dubbo | 服务治理 | 监控中心协议，如果为protocol="registry"，表示从注册中心发现监控中心地址，否则直连监控中心。 | 2.0.9以上版本 |
| address | <url> | string | 可选 | N/A | 服务治理 | 直连监控中心服务器地址，address="10.20.130.230:12080" | 1.0.16以上版本 |

#### Dubbo:application

应用信息配置。对应的配置类：com.alibaba.dubbo.config.ApplicationConfig

#### Dubbo:module

模块信息配置。对应的配置类 com.alibaba.dubbo.config.ModuleConfig

#### Dubbo:provider

服务提供者缺省值配置。对应的配置类： com.alibaba.dubbo.config.ProviderConfig。同时该标签为<dubbo:service> 和 <dubbo:protocol> 标签的缺省值设置。

#### Dubbo:consumer

服务消费者缺省值配置。配置类： com.alibaba.dubbo.config.ConsumerConfig 。同时该标签为<dubbo:reference> 标签的缺省值设置。

#### Dubbo:method

方法级配置。对应的配置类： com.alibaba.dubbo.config.MethodConfig。同时该标签为<dubbo:service> 或 <dubbo:reference> 的子标签，用于控制到方法级。

#### Dubbo:argument

方法参数配置。对应的配置类： com.alibaba.dubbo.config.ArgumentConfig。该标签为 <dubbo:method>的子标签，用于方法参数的特征描述，比如：

<dubbo:method name="findXxx" timeout="3000" retries="2">

<dubbo:argument index="0" callback="true" />

</dubbo:method>

#### Dubbo:parameter

选项参数配置。对应的配置类：java.util.Map。同时该标签为<dubbo:protocol>或<dubbo:service>或<dubbo:provider>或<dubbo:reference>或<dubbo:consumer>的子标签，用于配置自定义参数，该配置项将作为扩展点设置自定义参数使用。

| **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| key | key | string | **必填** |  | 服务治理 | 路由参数键 | 2.0.0以上版本 |
| value | value | string | **必填** |  | 服务治理 | 路由参数值 | 2.0.0以上版本 |

比如：

<dubbo:protocol name="napoli">

<dubbo:parameter key="http://10.20.160.198/wiki/display/dubbo/napoli.queue.name" value="xxx" />

</dubbo:protocol>

也可以：

<dubbo:protocol name="jms" p:queue="xxx" />

### 协议参考手册

#### dubbo://

Dubbo 缺省协议采用单一长连接和 NIO 异步通讯，适合于小数据量大并发的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况。

反之，Dubbo 缺省协议不适合传送大数据量的服务，比如传文件，传视频等，除非请求量很低。



* Transporter: mina, netty, grizzy
* Serialization: dubbo, hessian2, java, json
* Dispatcher: all, direct, message, execution, connection
* ThreadPool: fixed, cached

特性

缺省协议，使用基于 mina 1.1.7 和 hessian 3.2.1 的 tbremoting 交互。

* 连接个数：单连接
* 连接方式：长连接
* 传输协议：TCP
* 传输方式：NIO 异步传输
* 序列化：Hessian 二进制序列化
* 适用范围：传入传出参数数据包较小（建议小于100K），消费者比提供者个数多，单一消费者无法压满提供者，尽量不要用 dubbo 协议传输大文件或超大字符串。
* 适用场景：常规远程服务方法调用

约束

* 参数及返回值需实现 Serializable 接口
* 参数及返回值不能自定义实现 List, Map, Number, Date, Calendar 等接口，只能用 JDK 自带的实现，因为 hessian 会做特殊处理，自定义实现类中的属性值都会丢失。
* Hessian 序列化，只传成员属性值和值的类型，不传方法或静态变量，兼容情况 [[1]](http://dubbo.apache.org/zh-cn/docs/user/references/protocol/dubbo.html#fn1)[[2]](http://dubbo.apache.org/zh-cn/docs/user/references/protocol/dubbo.html#fn2)：

| **数据通讯** | **情况** | **结果** |
| --- | --- | --- |
| A->B | 类A多一种 属性（或者说类B少一种 属性） | 不抛异常，A多的那 个属性的值，B没有， 其他正常 |
| A->B | 枚举A多一种 枚举（或者说B少一种 枚举），A使用多 出来的枚举进行传输 | 抛异常 |
| A->B | 枚举A多一种 枚举（或者说B少一种 枚举），A不使用 多出来的枚举进行传输 | 不抛异常，B正常接 收数据 |
| A->B | A和B的属性 名相同，但类型不相同 | 抛异常 |
| A->B | serialId 不相同 | 正常传输 |

接口增加方法，对客户端无影响，如果该方法不是客户端需要的，客户端不需要重新部署。输入参数和结果集中增加属性，对客户端无影响，如果客户端并不需要新属性，不用重新部署。

输入参数和结果集属性名变化，对客户端序列化无影响，但是如果客户端不重新部署，不管输入还是输出，属性名变化的属性值是获取不到的。

总结：服务器端和客户端对领域对象并不需要完全一致，而是按照最大匹配原则。

配置

配置协议：

<dubbo:protocol name="dubbo" port="20880" />

设置默认协议：

<dubbo:provider protocol="dubbo" />

设置服务协议：

<dubbo:service protocol="dubbo" />

多端口：

<dubbo:protocol id="dubbo1" name="dubbo" port="20880" />

<dubbo:protocol id="dubbo2" name="dubbo" port="20881" />

配置协议选项：

<dubbo:protocol name=“dubbo” port=“9090” server=“netty” client=“netty” codec=“dubbo” serialization=“hessian2” charset=“UTF-8” threadpool=“fixed” threads=“100” queues=“0” iothreads=“9” buffer=“8192” accepts=“1000” payload=“8388608” />

多连接配置：

Dubbo 协议缺省每服务每提供者每消费者使用单一长连接，如果数据量较大，可以使用多个连接。

<dubbo:service connections="1"/>

<dubbo:reference connections="1"/>

* <dubbo:service connections="0"> 或 <dubbo:reference connections="0"> 表示该服务使用 JVM 共享长连接。**缺省**
* <dubbo:service connections="1"> 或 <dubbo:reference connections="1"> 表示该服务使用独立长连接。
* <dubbo:service connections="2"> 或<dubbo:reference connections="2"> 表示该服务使用独立两条长连接。

为防止被大量连接撑挂，可在服务提供方限制大接收连接数，以实现服务提供方自我保护。

<dubbo:protocol name="dubbo" accepts="1000" />

dubbo.properties 配置：

dubbo.service.protocol=dubbo

常见问题

为什么要消费者比提供者个数多?

因 dubbo 协议采用单一长连接，假设网络为千兆网卡 [[3]](http://dubbo.apache.org/zh-cn/docs/user/references/protocol/dubbo.html#fn3)，根据测试经验数据每条连接最多只能压满 7MByte(不同的环境可能不一样，供参考)，理论上 1 个服务提供者需要 20 个服务消费者才能压满网卡。

为什么不能传大包?

因 dubbo 协议采用单一长连接，如果每次请求的数据包大小为 500KByte，假设网络为千兆网卡 [[3:1]](http://dubbo.apache.org/zh-cn/docs/user/references/protocol/dubbo.html#fn3)，每条连接最大 7MByte(不同的环境可能不一样，供参考)，单个服务提供者的 TPS(每秒处理事务数)最大为：128MByte / 500KByte = 262。单个消费者调用单个服务提供者的 TPS(每秒处理事务数)最大为：7MByte / 500KByte = 14。如果能接受，可以考虑使用，否则网络将成为瓶颈。

为什么采用异步单一长连接?

因为服务的现状大都是服务提供者少，通常只有几台机器，而服务的消费者多，可能整个网站都在访问该服务，比如 Morgan 的提供者只有 6 台提供者，却有上百台消费者，每天有 1.5 亿次调用，如果采用常规的 hessian 服务，服务提供者很容易就被压跨，通过单一连接，保证单一消费者不会压死提供者，长连接，减少连接握手验证等，并使用异步 IO，复用线程池，防止 C10K 问题。

#### rmi://

RMI 协议采用 JDK 标准的 java.rmi.\* 实现，采用阻塞式短连接和 JDK 标准序列化方式。

注意：如果正在使用 RMI 提供服务给外部访问 [[1]](http://dubbo.apache.org/zh-cn/docs/user/references/protocol/rmi.html#fn1)，同时应用里依赖了老的 common-collections 包 [[2]](http://dubbo.apache.org/zh-cn/docs/user/references/protocol/rmi.html#fn2) 的情况下，存在反序列化安全风险 [[3]](http://dubbo.apache.org/zh-cn/docs/user/references/protocol/rmi.html#fn3)。

特性

* 连接个数：多连接
* 连接方式：短连接
* 传输协议：TCP
* 传输方式：同步传输
* 序列化：Java 标准二进制序列化
* 适用范围：传入传出参数数据包大小混合，消费者与提供者个数差不多，可传文件。
* 适用场景：常规远程服务方法调用，与原生RMI服务互操作

约束

* 参数及返回值需实现 Serializable 接口
* dubbo 配置中的超时时间对 RMI 无效，需使用 java 启动参数设置：-Dsun.rmi.transport.tcp.responseTimeout=3000，参见下面的 RMI 配置

dubbo.properties 配置

dubbo.service.protocol=rmi

RMI配置

java -Dsun.rmi.transport.tcp.responseTimeout=3000

更多 RMI 优化参数请查看 [JDK 文档](http://download.oracle.com/docs/cd/E17409_01/javase/6/docs/technotes/guides/rmi/sunrmiproperties.html)

接口

如果服务接口继承了 java.rmi.Remote 接口，可以和原生 RMI 互操作，即：

* 提供者用 Dubbo 的 RMI 协议暴露服务，消费者直接用标准 RMI 接口调用，
* 或者提供方用标准 RMI 暴露服务，消费方用 Dubbo 的 RMI 协议调用。

如果服务接口没有继承 java.rmi.Remote 接口：

* 缺省 Dubbo 将自动生成一个 com.xxx.XxxService$Remote 的接口，并继承 java.rmi.Remote 接口，并以此接口暴露服务，
* 但如果设置了 <dubbo:protocol name="rmi" codec="spring" />，将不生成 $Remote 接口，而使用 Spring 的 RmiInvocationHandler 接口暴露服务，和 Spring 兼容。

配置

定义 RMI 协议：

<dubbo:protocol name="rmi" port="1099" />

设置默认协议：

<dubbo:provider protocol="rmi" />

设置服务协议：

<dubbo:service protocol="rmi" />

多端口：

<dubbo:protocol id="rmi1" name="rmi" port="1099" />

<dubbo:protocol id="rmi2" name="rmi" port="2099" />

<dubbo:service protocol="rmi1" />

Spring 兼容性：

<dubbo:protocol name="rmi" codec="spring" />

#### hessian://

Hessian [[1]](http://dubbo.apache.org/zh-cn/docs/user/references/protocol/hessian.html#fn1) 协议用于集成 Hessian 的服务，Hessian 底层采用 Http 通讯，采用 Servlet 暴露服务，Dubbo 缺省内嵌 Jetty 作为服务器实现。

Dubbo 的 Hessian 协议可以和原生 Hessian 服务互操作，即：

* 提供者用 Dubbo 的 Hessian 协议暴露服务，消费者直接用标准 Hessian 接口调用
* 或者提供方用标准 Hessian 暴露服务，消费方用 Dubbo 的 Hessian 协议调用。

特性

* 连接个数：多连接
* 连接方式：短连接
* 传输协议：HTTP
* 传输方式：同步传输
* 序列化：Hessian二进制序列化
* 适用范围：传入传出参数数据包较大，提供者比消费者个数多，提供者压力较大，可传文件。
* 适用场景：页面传输，文件传输，或与原生hessian服务互操作

依赖

<dependency>

<groupId>com.caucho</groupId>

<artifactId>hessian</artifactId>

<version>4.0.7</version>

</dependency>

约束

* 参数及返回值需实现 Serializable 接口
* 参数及返回值不能自定义实现 List, Map, Number, Date, Calendar 等接口，只能用 JDK 自带的实现，因为 hessian 会做特殊处理，自定义实现类中的属性值都会丢失。

配置

定义 hessian 协议：

<dubbo:protocol name="hessian" port="8080" server="jetty" />

设置默认协议：

<dubbo:provider protocol="hessian" />

设置 service 协议：

<dubbo:service protocol="hessian" />

多端口：

<dubbo:protocol id="hessian1" name="hessian" port="8080" />

<dubbo:protocol id="hessian2" name="hessian" port="8081" />

直连：

<dubbo:reference id="helloService" interface="HelloWorld" url="hessian://10.20.153.10:8080/helloWorld" />

1. [Hessian](http://hessian.caucho.com/) 是 Caucho 开源的一个 RPC 框架，其通讯效率高于 WebService 和 Java 自带的序列化。 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/references/protocol/hessian.html#fnref1)

#### http://

基于 HTTP 表单的远程调用协议，采用 Spring 的 HttpInvoker 实现 [[1]](http://dubbo.apache.org/zh-cn/docs/user/references/protocol/http.html#fn1)

特性

* 连接个数：多连接
* 连接方式：短连接
* 传输协议：HTTP
* 传输方式：同步传输
* 序列化：表单序列化
* 适用范围：传入传出参数数据包大小混合，提供者比消费者个数多，可用浏览器查看，可用表单或URL传入参数，暂不支持传文件。
* 适用场景：需同时给应用程序和浏览器 JS 使用的服务。

约束

* 参数及返回值需符合 Bean 规范

配置

配置协议：

<dubbo:protocol name="http" port="8080" />

配置 Jetty Server (默认)：

<dubbo:protocol ... server="jetty" />

配置 Servlet Bridge Server (推荐使用)：

<dubbo:protocol ... server="servlet" />

配置 DispatcherServlet：

<servlet>

<servlet-name>dubbo</servlet-name>

<servlet-class>com.alibaba.dubbo.remoting.http.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dubbo</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

注意，如果使用 servlet 派发请求：

* 协议的端口 <dubbo:protocol port="8080" /> 必须与 servlet 容器的端口相同，
* 协议的上下文路径 <dubbo:protocol contextpath="foo" /> 必须与 servlet 应用的上下文路径相同。

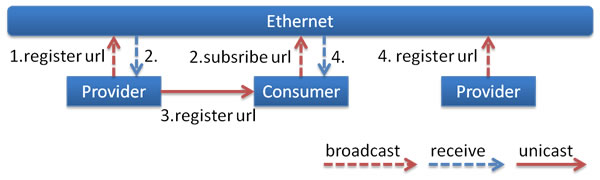
1. 2.3.0 以上版本支持 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/references/protocol/http.html#fnref1)

### 注册中心手册

推荐使用zookeeper注册中心

#### Multicase注册中心

Multicast 注册中心不需要启动任何中心节点，只要广播地址一样，就可以互相发现。



1. 提供方启动时广播自己的地址
2. 消费方启动时广播订阅请求
3. 提供方收到订阅请求时，单播自己的地址给订阅者，如果设置了 unicast=false，则广播给订阅者
4. 消费方收到提供方地址时，连接该地址进行 RPC 调用。

组播受网络结构限制，只适合小规模应用或开发阶段使用。组播地址段: 224.0.0.0 - 239.255.255.255

配置

<dubbo:registry address="multicast://224.5.6.7:1234" />

或

<dubbo:registry protocol="multicast" address="224.5.6.7:1234" />

为了减少广播量，Dubbo 缺省使用单播发送提供者地址信息给消费者，如果一个机器上同时启了多个消费者进程，消费者需声明 unicast=false，否则只会有一个消费者能收到消息：

<dubbo:registry address="multicast://224.5.6.7:1234?unicast=false" />

或

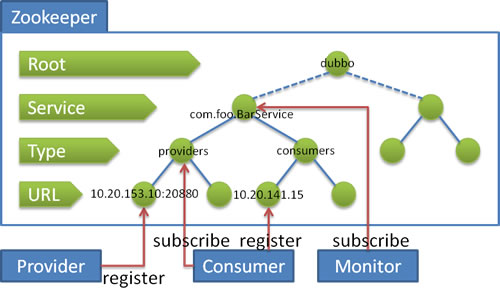
<dubbo:registry protocol="multicast" address="224.5.6.7:1234">

<dubbo:parameter key="unicast" value="false" />

</dubbo:registry>

#### Zookeeper注册中心

[Zookeeper](http://zookeeper.apache.org/) 是 Apacahe Hadoop 的子项目，是一个树型的目录服务，支持变更推送，适合作为 Dubbo 服务的注册中心，工业强度较高，可用于生产环境，并推荐使用 [[1]](http://dubbo.apache.org/zh-cn/docs/user/references/registry/zookeeper.html#fn1)。



流程说明：

* 服务提供者启动时: 向 /dubbo/com.foo.BarService/providers 目录下写入自己的 URL 地址
* 服务消费者启动时: 订阅 /dubbo/com.foo.BarService/providers 目录下的提供者 URL 地址。并向/dubbo/com.foo.BarService/consumers 目录下写入自己的 URL 地址
* 监控中心启动时: 订阅 /dubbo/com.foo.BarService 目录下的所有提供者和消费者 URL 地址。

支持以下功能：

* 当提供者出现断电等异常停机时，注册中心能自动删除提供者信息
* 当注册中心重启时，能自动恢复注册数据，以及订阅请求
* 当会话过期时，能自动恢复注册数据，以及订阅请求
* 当设置 <dubbo:registry check="false" /> 时，记录失败注册和订阅请求，后台定时重试
* 可通过 <dubbo:registry username="admin" password="1234" /> 设置 zookeeper 登录信息
* 可通过 <dubbo:registry group="dubbo" /> 设置 zookeeper 的根节点，不设置将使用无根树
* 支持 \* 号通配符 <dubbo:reference group="\*" version="\*" />，可订阅服务的所有分组和所有版本的提供者

使用

在 provider 和 consumer 中增加 zookeeper 客户端 jar 包依赖：

<dependency>

<groupId>org.apache.zookeeper</groupId>

<artifactId>zookeeper</artifactId>

<version>3.3.3</version>

</dependency>

或直接[下载](http://repo1.maven.org/maven2/org/apache/zookeeper/zookeeper)。

Dubbo 支持 zkclient 和 curator 两种 Zookeeper 客户端实现：

使用 zkclient 客户端

从 2.2.0 版本开始缺省为 zkclient 实现，以提升 zookeeper 客户端的健状性。[zkclient](https://github.com/sgroschupf/zkclient) 是 Datameer 开源的一个 Zookeeper 客户端实现。

缺省配置：

<dubbo:registry ... client="zkclient" />

或：

dubbo.registry.client=zkclient

或：

zookeeper://10.20.153.10:2181?client=zkclient

需依赖或直接[下载](http://repo1.maven.org/maven2/com/github/sgroschupf/zkclient)：

<dependency>

<groupId>com.github.sgroschupf</groupId>

<artifactId>zkclient</artifactId>

<version>0.1</version>

</dependency>

使用 curator 客户端

从 2.3.0 版本开始支持可选 curator 实现。[Curator](https://github.com/Netflix/curator) 是 Netflix 开源的一个 Zookeeper 客户端实现。

如果需要改为 curator 实现，请配置：

<dubbo:registry ... client="curator" />

或：

dubbo.registry.client=curator

或：

zookeeper://10.20.153.10:2181?client=curator

需依赖或直接[下载](http://repo1.maven.org/maven2/com/netflix/curator/curator-framework)：

<dependency>

<groupId>com.netflix.curator</groupId>

<artifactId>curator-framework</artifactId>

<version>1.1.10</version>

</dependency>

Zookeeper 单机配置:

<dubbo:registry address="zookeeper://10.20.153.10:2181" />

或：

<dubbo:registry protocol="zookeeper" address="10.20.153.10:2181" />

Zookeeper 集群配置：

<dubbo:registry address="zookeeper://10.20.153.10:2181?backup=10.20.153.11:2181,10.20.153.12:2181" />

或：

<dubbo:registry protocol="zookeeper" address="10.20.153.10:2181,10.20.153.11:2181,10.20.153.12:2181" />

同一 Zookeeper，分成多组注册中心:

<dubbo:registry id="chinaRegistry" protocol="zookeeper" address="10.20.153.10:2181" group="china" />

<dubbo:registry id="intlRegistry" protocol="zookeeper" address="10.20.153.10:2181" group="intl" />

zookeeper 安装

安装方式参见: [Zookeeper安装手册](http://dubbo.apache.org/books/dubbo-admin-book/install/zookeeper.html)，只需搭一个原生的 Zookeeper 服务器，并将 [Quick Start](http://dubbo.apache.org/zh-cn/docs/user/preface/usage.html) 中 Provider 和 Consumer 里的 conf/dubbo.properties 中的 dubbo.registry.addrss 的值改为zookeeper://127.0.0.1:2181 即可使用。

可靠性声明

阿里内部并没有采用 Zookeeper 做为注册中心，而是使用自己实现的基于数据库的注册中心，即：Zookeeper 注册中心并没有在阿里内部长时间运行的可靠性保障，此 Zookeeper 桥接实现只为开源版本提供，其可靠性依赖于 Zookeeper 本身的可靠性。

兼容性声明

因 2.0.8 最初设计的 zookeeper 存储结构不能扩充不同类型的数据，2.0.9 版本做了调整，所以不兼容，需全部改用 2.0.9 版本才行，以后的版本会保持兼容 2.0.9。2.2.0 版本改为基于 zkclient 实现，需增加 zkclient 的依赖包，2.3.0 版本增加了基于 curator 的实现，作为可选实现策略。

#### Redis注册中心

#### Simple注册中心

### Telnet命令参考手册

从 2.0.5 版本开始，dubbo 开始支持通过 telnet 命令来进行服务治理。

使用

telnet localhost 20880

或者：

echo status | nc -i 1 localhost 20880

status命令所检查的资源也可以扩展，参见：[扩展参考手册](http://dubbo.apache.org/zh-cn/docs/dev/impls/status-checker.html)。

命令

以下展示了 dubbo 内建的 telnet 命令的说明和用法，此外，telnet 命令还支持用户自行扩展，参见：[Telnet 命令扩展](http://dubbo.apache.org/zh-cn/docs/dev/impls/telnet-handler.html)。

ls

1. ls: 显示服务列表
2. ls -l: 显示服务详细信息列表
3. ls XxxService: 显示服务的方法列表
4. ls -l XxxService: 显示服务的方法详细信息列表

ps

1. ps: 显示服务端口列表
2. ps -l: 显示服务地址列表
3. ps 20880: 显示端口上的连接信息
4. ps -l 20880: 显示端口上的连接详细信息

cd

1. cd XxxService: 改变缺省服务，当设置了缺省服务，凡是需要输入服务名作为参数的命令，都可以省略服务参数
2. cd /: 取消缺省服务

pwd

pwd: 显示当前缺省服务

trace

1. trace XxxService: 跟踪 1 次服务任意方法的调用情况
2. trace XxxService 10: 跟踪 10 次服务任意方法的调用情况
3. trace XxxService xxxMethod: 跟踪 1 次服务方法的调用情况
4. trace XxxService xxxMethod 10: 跟踪 10 次服务方法的调用情况

count

1. count XxxService: 统计 1 次服务任意方法的调用情况
2. count XxxService 10: 统计 10 次服务任意方法的调用情况
3. count XxxService xxxMethod: 统计 1 次服务方法的调用情况
4. count XxxService xxxMethod 10: 统计 10 次服务方法的调用情况

invoke

1. invoke XxxService.xxxMethod({"prop": "value"}): 调用服务的方法
2. invoke xxxMethod({"prop": "value"}): 调用服务的方法(自动查找包含此方法的服务)

status

1. status: 显示汇总状态，该状态将汇总所有资源的状态，当全部 OK 时则显示 OK，只要有一个 ERROR 则显示 ERROR，只要有一个 WARN 则显示 WARN
2. status -l: 显示状态列表

log [[1]](http://dubbo.apache.org/zh-cn/docs/user/references/telnet.html#fn1)

1. log debug: 修改 dubbo logger 的日志级别
2. log 100: 查看 file logger 的最后 100 字符的日志

help

1. help: 显示 telnet 命帮助信息
2. help xxx: 显示xxx命令的详细帮助信息

clear

1. clear: 清除屏幕上的内容
2. clear 100: 清除屏幕上的指定行数的内容

exit

exit: 退出当前 telnet 命令行

相关参数说明

QoS提供了一些启动参数，来对启动进行配置，他们主要包括：

| **参数** | **说明** | **默认值** |
| --- | --- | --- |
| qosEnable | 是否启动QoS | true |
| qosPort | 启动QoS绑定的端口 | 22222 |
| qosAcceptForeignIp | 是否允许远程访问 | false |

注意，从2.6.4/2.7.0开始，qosAcceptForeignIp默认配置改为false，如果qosAcceptForeignIp设置为true，有可能带来安全风险，请仔细评估后再打开。

QoS参数可以通过如下方式进行配置

* 系统属性
* dubbo.properties
* XML方式
* Spring-boot自动装配方式

其中，上述方式的优先顺序为系统属性 > dubbo.properties > XML/Spring-boot自动装配方式。

使用系统属性方式进行配置

-Ddubbo.application.qos.enable=true

-Ddubbo.application.qos.port=33333

-Ddubbo.application.qos.accept.foreign.ip=false

使用dubbo.properties文件进行配置

在项目的src/main/resources目录下添加dubbo.properties文件，内容如下:

dubbo.application.qos.enable=true

dubbo.application.qos.port=33333

dubbo.application.qos.accept.foreign.ip=false

使用XML方法进行配置

如果要通过XML配置响应的QoS相关的参数，可以进行如下配置：

**<?**xml version="1.0" encoding="UTF-8"**?>**

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

<dubbo:application name="demo-provider">

<dubbo:parameter key="qos.enable" value="true"/>

<dubbo:parameter key="qos.accept.foreign.ip" value="false"/>

<dubbo:parameter key="qos.port" value="33333"/>

</dubbo:application>

<dubbo:registry address="multicast://224.5.6.7:1234"/>

<dubbo:protocol name="dubbo" port="20880"/>

<dubbo:service interface="org.apache.dubbo.demo.provider.DemoService" ref="demoService"/>

<bean id="demoService" class="org.apache.dubbo.demo.provider.DemoServiceImpl"/>

</beans>

使用spring-boot自动装配方式配置

如果是spring-boot的应用，可以在application.properties或者application.yml上配置:

dubbo.application.qosEnable=true

dubbo.application.qosPort=33333

dubbo.application.qosAcceptForeignIp=false

1. 2.0.6 以上版本支持 [↩︎](http://dubbo.apache.org/zh-cn/docs/user/references/telnet.html#fnref1)

## 注册中心：zookeeper

## 配置中心：Apollo

# 信息队列

## activeMq

参考文档：<https://www.jianshu.com/p/8caa6d66b10d>

### 概述与介绍

ActiveMQ是Apache出品，最流行的，功能最强的即时通讯和和集成模式的开源服务器。

ActiveMQ是一个完全支持JMS1.1和J2EE1.4规范的JMS Provider实现。

提供客户端支持跨语言和协议，带有易于在充分支持JMS 1.1和1.4使用J2EE企业集成模式和许多先进的功能。

### 特性

1、多种语言和协议编写客户端。语言：java，C，C++，C#，Python，Perl. Python. PHP。应用协议：OpenWire. Stomp REST. WS Notification. XMPP. AMQP。

2、完全支持JMS1.1和J2EE1.4规范（持久化，XA消息，事务)

3、对Spring的支持，ActiveMQ可以很容易内嵌到使用Spring的系统里面去，而且也支持Spring2.0的特性

4、通过了常见J2EE服务器（如 Geronimo. JBoss 4. GlassFish. WebLogic)的测试，其中通过JCA 1.5 resource adaptors的配置，可以让ActiveMQ可以自动的部署到任何兼容J2EE 1.4 商业服务器上

5、支持多种传送协议：in-VM. TCP. SSL. NIO. UDP. JGroups. JXTA

6、支持通过JDBC和journal提供高速的消息持久化

7、从设计上保证了高性能的集群，客户端-服务器，点对点

8、支持ajax

9、支持与axis整合

10、可以很容易得调用内嵌JMS provider，进行测试

### 安装ActiveMQ

1、下载activeMQ压缩包，解压

2、进入bin，根据电脑位数今晚win32或win64的目录，执行activemq.bat

3、默认管理界面url：<http://localhost:8161/>

4、默认服务的地址：tcp://localhost:61616

### 应用

#### ActiveMQ消息有三种形式

|  |  |  |
| --- | --- | --- |
| **JMS公共** | **点对点域** | **发布/订阅域** |
| ConnectionFactory | QueueConnectionFactory | TopicConnectionFactory |
| Connection | QueueConnection | TopicConnection |
| Destination | Queue | Topic |
| Session | QueueSession | TopicSession |
| MessageProducer | QueueSender | TopicPublisher |
| MessageConsumer | QueueReceiver | TopicSubscriber |

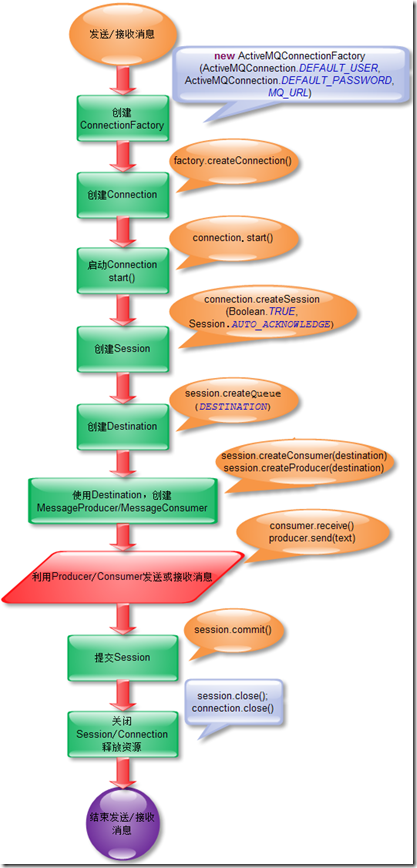
1、点对点方式（point-to-point）

点对点的消息发送方式主要建立在 Message Queue,Sender,reciever上，Message Queue 存贮消息，Sneder 发送消息，receive接收消息.具体点就是Sender Client发送Message Queue ,而 receiver Cliernt从Queue中接收消息和"发送消息已接受"到Quere,确认消息接收。消息发送客户端与接收客户端没有时间上的依赖，发送客户端可以在任何时刻发送信息到Queue，而不需要知道接收客户端是不是在运行

2、发布/订阅方式（publish/subscriber Messaging）

发布/订阅方式用于多接收客户端的方式.作为发布订阅的方式，可能存在多个接收客户端，并且接收端客户端与发送客户端存在时间上的依赖。一个接收端只能接收他创建以后发送客户端发送的信息。作为subscriber ,在接收消息时有两种方法，destination的receive方法，和实现message listener 接口的onMessage 方法。

#### 发送和接收信息基本流程



**发送消息的基本步骤：**

(1)、创建连接使用的工厂类JMS ConnectionFactory

(2)、使用管理对象JMS ConnectionFactory建立连接Connection，并启动

(3)、使用连接Connection 建立会话Session

(4)、使用会话Session和管理对象Destination创建消息生产者MessageSender

(5)、使用消息生产者MessageSender发送消息

**消息接收者从JMS接受消息的步骤**

(1)、创建连接使用的工厂类JMS ConnectionFactory

(2)、使用管理对象JMS ConnectionFactory建立连接Connection，并启动

(3)、使用连接Connection 建立会话Session

(4)、使用会话Session和管理对象Destination创建消息接收者MessageReceiver

(5)、使用消息接收者MessageReceiver接受消息，需要用setMessageListener将MessageListener接口绑定到MessageReceiver消息接收者必须实现了MessageListener接口，需要定义onMessage事件方法。

#### 示例代码

##### MessageSender

|  |
| --- |
| package org.xzh.springboot.activemq;  import javax.jms.Connection;  import javax.jms.ConnectionFactory;  import javax.jms.DeliveryMode;  import javax.jms.Destination;  import javax.jms.JMSException;  import javax.jms.MessageProducer;  import javax.jms.Session;  import javax.jms.TextMessage;  import org.apache.activemq.ActiveMQConnection;  import org.apache.activemq.ActiveMQConnectionFactory;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  public class MessageSender {    private static final Logger logger = LoggerFactory.getLogger(MessageSender.class);  /\*\*  \* 发送次数  \*/  public static final Integer SEND\_NUM = 5;    public static final String BROKER\_URL = "tcp://localhost:61616";    public static final String DESTINATION = "segde.mq.queue";    public void sendMessage(Session session, MessageProducer producer) throws JMSException{  for(int i=0; i<SEND\_NUM; i++){  String text = "发送第"+i+"条记录";  TextMessage textMessage = session.createTextMessage(text);  producer.send(textMessage);  logger.info("send: {}", text);  }  }    public void run() throws JMSException{  Connection conn = null;  Session session = null;  try{  // 创建链接工厂  ConnectionFactory factory = new ActiveMQConnectionFactory(ActiveMQConnection.DEFAULT\_USER,  ActiveMQConnection.DEFAULT\_PASSWORD, BROKER\_URL);  // 通过工厂创建一个连接  conn = factory.createConnection();  // 启动连接  conn.start();  // 创建一个session会话  session = conn.createSession(Boolean.TRUE, Session.AUTO\_ACKNOWLEDGE);  // 创建一个消息队列  Destination destination = session.createQueue(DESTINATION);  // 创建消息制作者  MessageProducer producer = session.createProducer(destination);  // 设置持久化模式  producer.setDeliveryMode(DeliveryMode.NON\_PERSISTENT);    //发送消息  sendMessage(session, producer);    // 提交会话  session.commit();  }catch(Exception e){  throw e;  }finally{  if(session != null){  session.close();  }  if(conn != null){  conn.close();  }  }  }    public static void main(String[] args) throws JMSException {  MessageSender sender = new MessageSender();  sender.run();  }  } |

##### MessageReceiver

|  |
| --- |
| package org.xzh.springboot.activemq;  import javax.jms.Connection;  import javax.jms.ConnectionFactory;  import javax.jms.Destination;  import javax.jms.JMSException;  import javax.jms.Message;  import javax.jms.MessageConsumer;  import javax.jms.Session;  import javax.jms.TextMessage;  import org.apache.activemq.ActiveMQConnection;  import org.apache.activemq.ActiveMQConnectionFactory;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  public class MessageReceiver {    private static final Logger logger = LoggerFactory.getLogger(MessageReceiver.class);  // tcp 地址  public static final String BROKER\_URL = "tcp://localhost:61616";    // 目标，在ActiveMQ管理员控制台创建 http://localhost:8161/admin/queues.jsp  public static final String DESTINATION = "segde.mq.queue";    public void run() throws JMSException{  Connection conn = null;  Session session = null;    try{  // 创建链接工厂  ConnectionFactory factory = new ActiveMQConnectionFactory(ActiveMQConnection.DEFAULT\_USER,  ActiveMQConnection.DEFAULT\_PASSWORD, BROKER\_URL);  // 通过工厂创建一个连接  conn = factory.createConnection();  // 启动连接  conn.start();  // 创建一个session会话  session = conn.createSession(Boolean.TRUE, Session.AUTO\_ACKNOWLEDGE);  // 创建一个消息队列  Destination destination = session.createQueue(DESTINATION);  // 创建消息制作者  MessageConsumer consumer = session.createConsumer(destination);    while(true){  // 接收数据的时间（等待） 100 ms  Message message = consumer.receive(1000\*100);    TextMessage textMessage = (TextMessage) message;  if(textMessage != null){  logger.info(textMessage.getText());  }  }  }catch(Exception e){  throw e;  }finally{  // 关闭释放资源  if(conn != null){  conn.close();  }    if(session != null){  session.close();  }  }  }    public static void main(String[] args) throws JMSException {  MessageReceiver receiver = new MessageReceiver();  receiver.run();  }  } |

##### QueueSender

|  |
| --- |
| package org.xzh.springboot.activemq;  import javax.jms.DeliveryMode;  import javax.jms.JMSException;  import javax.jms.MapMessage;  import javax.jms.Queue;  import javax.jms.QueueConnection;  import javax.jms.QueueConnectionFactory;  import javax.jms.QueueSession;  import javax.jms.Session;  import org.apache.activemq.ActiveMQConnection;  import org.apache.activemq.ActiveMQConnectionFactory;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  public class QueueSender {  private static final Logger logger = LoggerFactory.getLogger(QueueSender.class);    private static final String BROKER\_URL = "tcp://localhost:61616";    private static final String DESTATION = "queue.mq.msg";    private void sender(QueueSession session, javax.jms.QueueSender sender) throws JMSException{  for(int i=0; i<5; i++){  String text = "hhhh"+i;  MapMessage mapMessage = session.createMapMessage();  mapMessage.setString("text", text);  mapMessage.setLong("time", System.currentTimeMillis());  sender.send(mapMessage);  logger.info(text);  }  }    public void run() throws JMSException{  QueueConnection conn = null;  QueueSession session = null;    try{  QueueConnectionFactory factory = new ActiveMQConnectionFactory(ActiveMQConnection.DEFAULT\_USER,  ActiveMQConnection.DEFAULT\_PASSWORD, BROKER\_URL);  conn = factory.createQueueConnection();  conn.start();    session = conn.createQueueSession(Boolean.TRUE, Session.AUTO\_ACKNOWLEDGE);    Queue queue = session.createQueue(DESTATION);    javax.jms.QueueSender sender = session.createSender(queue);  sender.setDeliveryMode(DeliveryMode.NON\_PERSISTENT);    sender(session, sender);    session.commit();  }catch(Exception e){  throw e;  }finally{  if(session != null){  session.close();  }  if(conn != null){  conn.close();  }  }  }    public static void main(String[] args) throws JMSException {  QueueSender sender = new QueueSender();  sender.run();  }  } |

##### QueueReceiver

|  |
| --- |
| package org.xzh.springboot.activemq;  import javax.jms.MapMessage;  import javax.jms.Message;  import javax.jms.MessageListener;  import javax.jms.Queue;  import javax.jms.QueueConnection;  import javax.jms.QueueConnectionFactory;  import javax.jms.QueueSession;  import javax.jms.Session;  import org.apache.activemq.ActiveMQConnection;  import org.apache.activemq.ActiveMQConnectionFactory;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  public class QueueReceiver {  private static final Logger logger = LoggerFactory.getLogger(QueueReceiver.class);    private static final String BROKER\_URL = "tcp://localhost:61616";    private static final String DESTATION = "queue.mq.msg";    public void run() throws Exception{  QueueConnection conn = null;  QueueSession session = null;    try{  QueueConnectionFactory factory = new ActiveMQConnectionFactory(ActiveMQConnection.DEFAULT\_USER,  ActiveMQConnection.DEFAULT\_PASSWORD, BROKER\_URL);  conn = factory.createQueueConnection();  conn.start();    session = conn.createQueueSession(Boolean.TRUE, Session.AUTO\_ACKNOWLEDGE);    Queue queue = session.createQueue(DESTATION);  javax.jms.QueueReceiver receiver = session.createReceiver(queue);    receiver.setMessageListener(new MessageListener() {    @Override  public void onMessage(Message paramMessage) {  if(paramMessage != null){  MapMessage mapMessage = (MapMessage) paramMessage;  try{  String text = mapMessage.getString("text");  Long time = mapMessage.getLong("time");  logger.info("text:{}, time:{}", text, time);  }catch(Exception e){  logger.error("error", e);  }  }  }  });    Thread.sleep(1000 \* 100);  session.commit();    }catch(Exception e){  throw e;  }finally{  if(session != null){  session.close();  }  if(conn != null){  conn.close();  }  }  }    public static void main(String[] args) throws Exception {  QueueReceiver receiver = new QueueReceiver();  receiver.run();  }  } |

##### TopicSender

|  |
| --- |
| **package** org.xzh.springboot.activemq;  **import** javax.jms.DeliveryMode;  **import** javax.jms.JMSException;  **import** javax.jms.MapMessage;  **import** javax.jms.Session;  **import** javax.jms.Topic;  **import** javax.jms.TopicConnection;  **import** javax.jms.TopicConnectionFactory;  **import** javax.jms.TopicPublisher;  **import** javax.jms.TopicSession;  **import** org.apache.activemq.ActiveMQConnection;  **import** org.apache.activemq.ActiveMQConnectionFactory;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** TopicSender {  **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(TopicSender.**class**);    **private** **static** **final** String ***BROKER\_URL*** = "tcp://localhost:61616";    **private** **static** **final** String ***TOPIC*** = "topic.activemq";    **private** **void** send(TopicSession session, TopicPublisher publisher) **throws** JMSException{  **for**(**int** i = 0; i < 5; i++){  MapMessage mapMessage = session.createMapMessage();  String text = "text"+i;  mapMessage.setString("text", text);  mapMessage.setLong("time", System.*currentTimeMillis*());  publisher.send(mapMessage);  ***logger***.info("text:{}", text);  }  }    **public** **void** run() **throws** JMSException{  TopicConnection conn = **null**;  TopicSession session = **null**;    **try**{  TopicConnectionFactory factory = **new** ActiveMQConnectionFactory(ActiveMQConnection.***DEFAULT\_USER***,  ActiveMQConnection.***DEFAULT\_PASSWORD***, ***BROKER\_URL***);    conn = factory.createTopicConnection();  conn.start();    session = conn.createTopicSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);    Topic topic = session.createTopic(***TOPIC***);    TopicPublisher publisher = session.createPublisher(topic);  publisher.setDeliveryMode(DeliveryMode.***NON\_PERSISTENT***);    send(session, publisher);    session.commit();  }**catch**(Exception e){  **throw** e;  }**finally**{  **if**(session != **null**){  session.close();  }    **if**(conn != **null**){  conn.close();  }  }  }    **public** **static** **void** main(String[] args) **throws** JMSException {  **new** TopicSender().run();  }  } |

##### TopicReceiver

|  |
| --- |
| **package** org.xzh.springboot.activemq;  **import** javax.jms.MapMessage;  **import** javax.jms.Message;  **import** javax.jms.MessageListener;  **import** javax.jms.Session;  **import** javax.jms.Topic;  **import** javax.jms.TopicConnection;  **import** javax.jms.TopicConnectionFactory;  **import** javax.jms.TopicSession;  **import** javax.jms.TopicSubscriber;  **import** org.apache.activemq.ActiveMQConnection;  **import** org.apache.activemq.ActiveMQConnectionFactory;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** TopicReceiver {  **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(TopicReceiver.**class**);    **private** **static** **final** String ***BROKER\_URL*** = "tcp://localhost:61616";    **private** **static** **final** String ***TOPIC*** = "topic.activemq";    **public** **void** run() **throws** Exception{  TopicConnection conn = **null**;  TopicSession session = **null**;    **try**{  TopicConnectionFactory factory = **new** ActiveMQConnectionFactory(ActiveMQConnection.***DEFAULT\_USER***,  ActiveMQConnection.***DEFAULT\_PASSWORD***, ***BROKER\_URL***);    conn = factory.createTopicConnection();  conn.start();    session = conn.createTopicSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);    Topic topic = session.createTopic(***TOPIC***);    TopicSubscriber subscriber = session.createSubscriber(topic);    subscriber.setMessageListener(**new** MessageListener() {    @Override  **public** **void** onMessage(Message arg0) {  **if**(arg0 != **null**){  MapMessage mapMessage = (MapMessage) arg0;  **try**{  String text = mapMessage.getString("text");  Long time = mapMessage.getLong("time");  ***logger***.info("text:{}, time:{}", text, time);  }**catch**(Exception e){  ***logger***.error("error ", e);  }  }  }  });    // 休眠100ms再关闭  Thread.*sleep*(1000 \* 100);    session.commit();  }**catch**(Exception e){  **throw** e;  }**finally**{  **if**(session != **null**){  session.close();  }    **if**(conn != **null**){  conn.close();  }  }  }    **public** **static** **void** main(String[] args) **throws** Exception {  **new** TopicReceiver().run();  }  } |

## kafka

## rocketMq

<https://www.jianshu.com/p/824066d70da8>

### rocketmq是什么

RcoketMQ 是一款低延迟、高可靠、可伸缩、易于使用的消息中间件。具有以下特性：

1. 支持发布/订阅（Pub/Sub）和点对点（P2P）消息模型
2. 在一个队列中可靠的先进先出（FIFO）和严格的顺序传递
3. 支持拉（pull）和推（push）两种消息模式
4. 单一队列百万消息的堆积能力
5. 支持多种消息协议，如 JMS、MQTT 等
6. 分布式高可用的部署架构,满足至少一次消息传递语义
7. 提供 docker 镜像用于隔离测试和云集群部署
8. 提供配置、指标和监控等功能丰富的 Dashboard

对于这些特性描述，大家简单过一眼就即可，深入学习之后自然就明白了。

### 专业术语

* Producer

消息生产者，生产者的作用就是将消息发送到 MQ，生产者本身既可以产生消息，如读取文本信息等。也可以对外提供接口，由外部应用来调用接口，再由生产者将收到的消息发送到 MQ。

* Producer Group

生产者组，简单来说就是多个发送同一类消息的生产者称之为一个生产者组。

* Consumer

消息消费者，简单来说，消费 MQ 上的消息的应用程序就是消费者，至于消息是否进行逻辑处理，还是直接存储到数据库等取决于业务需要。

* Consumer Group

消费者组，和生产者类似，消费同一类消息的多个 consumer 实例组成一个消费者组。

* Topic

Topic 是一种消息的逻辑分类，比如说你有订单类的消息，也有库存类的消息，那么就需要进行分类，一个是订单 Topic 存放订单相关的消息，一个是库存 Topic 存储库存相关的消息。

* Message

Message 是消息的载体。一个 Message 必须指定 topic，相当于寄信的地址。Message 还有一个可选的 tag 设置，以便消费端可以基于 tag 进行过滤消息。也可以添加额外的键值对，例如你需要一个业务 key 来查找 broker 上的消息，方便在开发过程中诊断问题。

* Tag

标签可以被认为是对 Topic 进一步细化。一般在相同业务模块中通过引入标签来标记不同用途的消息。

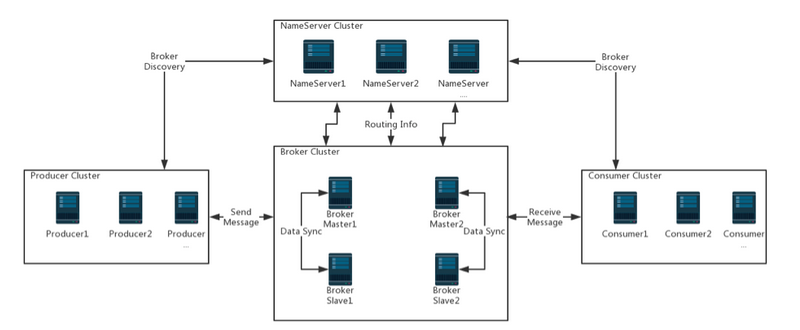
* Broker

Broker 是 RocketMQ 系统的主要角色，其实就是前面一直说的 MQ。Broker 接收来自生产者的消息，储存以及为消费者拉取消息的请求做好准备。

* Name Server

Name Server 为 producer 和 consumer 提供路由信息。

### Rocketmq架构



由这张图可以看到有四个集群，分别是 NameServer 集群、Broker 集群、Producer 集群和 Consumer 集群：

1. NameServer: 提供轻量级的服务发现和路由。 每个 NameServer 记录完整的路由信息，提供等效的读写服务，并支持快速存储扩展。
2. Broker: 通过提供轻量级的 Topic 和 Queue 机制来处理消息存储,同时支持推（push）和拉（pull）模式以及主从结构的容错机制。
3. Producer：生产者，产生消息的实例，拥有相同 Producer Group 的 Producer 组成一个集群。
4. Consumer：消费者，接收消息进行消费的实例，拥有相同 Consumer Group 的  
   Consumer 组成一个集群。

简单说明一下图中箭头含义，从 Broker 开始，Broker Master1 和 Broker Slave1 是主从结构，它们之间会进行数据同步，即 Date Sync。同时每个 Broker 与  
NameServer 集群中的所有节  
点建立长连接，定时注册 Topic 信息到所有 NameServer 中。

Producer 与 NameServer 集群中的其中一个节点（随机选择）建立长连接，定期从 NameServer 获取 Topic 路由信息，并向提供 Topic 服务的 Broker Master 建立长连接，且定时向 Broker 发送心跳。Producer 只能将消息发送到 Broker master，但是 Consumer 则不一样，它同时和提供 Topic 服务的 Master 和 Slave  
建立长连接，既可以从 Broker Master 订阅消息，也可以从 Broker Slave 订阅消息。

### Rocketmq集群部署模式

 单 master 模式

也就是只有一个 master 节点，称不上是集群，一旦这个 master 节点宕机，那么整个服务就不可用，适合个人学习使用。

 多 master 模式

多个 master 节点组成集群，单个 master 节点宕机或者重启对应用没有影响。  
优点：所有模式中性能最高  
缺点：单个 master 节点宕机期间，未被消费的消息在节点恢复之前不可用，消息的实时性就受到影响。

**注意**：使用同步刷盘可以保证消息不丢失，同时 Topic 相对应的 queue 应该分布在集群中各个节点，而不是只在某各节点上，否则，该节点宕机会对订阅该 topic 的应用造成影响。

 多 master 多 slave 异步复制模式

在多 master 模式的基础上，每个 master 节点都有至少一个对应的 slave。Master

节点可读可写，但是 slave 只能读不能写，类似于 mysql 的主备模式。  
优点： 在 master 宕机时，消费者可以从 slave 读取消息，消息的实时性不会受影响，性能几乎和多 master 一样。

缺点：使用异步复制的同步方式有可能会有消息丢失的问题。

 多 master 多 slave 同步双写模式

同多 master 多 slave 异步复制模式类似，区别在于 master 和 slave 之间的数据同步方式。

优点：同步双写的同步模式能保证数据不丢失。

缺点：发送单个消息 RT 会略长，性能相比异步复制低10%左右。

刷盘策略：同步刷盘和异步刷盘（指的是节点自身数据是同步还是异步存储）

同步方式：同步双写和异步复制（指的一组 master 和 slave 之间数据的同步）

**注意**：要保证数据可靠，需采用同步刷盘和同步双写的方式，但性能会较其他方式低。

### 开发

1、pom.xml引入

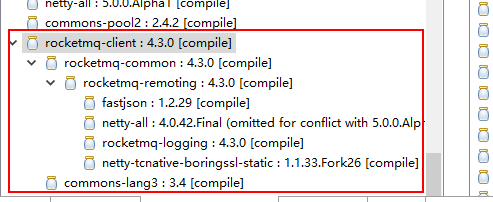
<dependency>

<groupId>org.apache.rocketmq</groupId>

<artifactId>rocketmq-client</artifactId>

<version>4.3.0</version>

</dependency>



2、producer

|  |
| --- |
| **public** **class** Producer {  **public** **static** **void** main(String[] args) **throws** MQClientException,  InterruptedException {  // 声明并初始化一个producer  // 需要一个producer group名字作为构造方法的参数，这里为producer\_test1  DefaultMQProducer producer = **new** DefaultMQProducer("producer\_test1");  // 设置NameServer地址,此处应改为实际NameServer地址，多个地址之间用;分隔  // NameServer的地址必须有，但是也可以通过环境变量的方式设置，不一定非得写死在代码里  producer.setNamesrvAddr("127.0.0.1:9876");// 127.0.0.1:9876;10.1.54.122:9876  // 调用start()方法启动一个producer实例  producer.start();  // 发送1条消息到Topic为Topic\_test1，tag为Tag\_test1，消息内容为Body\_test1  **try** {  Message msg = **new** Message("Topic\_test1","Tag\_test1",  "Body\_test1".getBytes(RemotingHelper.***DEFAULT\_CHARSET***));  // 调用producer的send()方法发送消息  // 这里调用的是同步的方式，所以会有返回结果  SendResult sendResult = producer.send(msg);  // 打印返回结果，可以看到消息发送的状态以及一些相关信息  System.***out***.println(sendResult);  } **catch** (Exception e) {  e.printStackTrace();  producer.shutdown();  }  // 发送完消息之后，调用shutdown()方法关闭producer  producer.shutdown();  }  } |

3、consumer

|  |
| --- |
| **public** **class** Consumer {  **public** **static** **void** main(String[] args) **throws** InterruptedException,  MQClientException {  // 声明并初始化一个consumer  // 需要一个consumer group名字作为构造方法的参数，这里为consumer1  DefaultMQPushConsumer consumer = **new** DefaultMQPushConsumer(  "consumer\_test1");  // 同样也要设置NameServer地址  consumer.setNamesrvAddr("127.0.0.1:9876");// 10.1.54.121:9876;10.1.54.122:9876  // 这里设置的是一个consumer的消费策略  // CONSUME\_FROM\_LAST\_OFFSET 默认策略，从该队列最尾开始消费，即跳过历史消息  // CONSUME\_FROM\_FIRST\_OFFSET 从队列最开始开始消费，即历史消息（还储存在broker的）全部消费一遍  // CONSUME\_FROM\_TIMESTAMP  // 从某个时间点开始消费，和setConsumeTimestamp()配合使用，默认是半个小时以前  consumer.setConsumeFromWhere(ConsumeFromWhere.***CONSUME\_FROM\_FIRST\_OFFSET***);  // 设置consumer所订阅的Topic和Tag，\*代表全部的Tag  consumer.subscribe("Topic\_test1", "\*");  // 设置一个Listener，主要进行消息的逻辑处理  consumer.registerMessageListener(**new** MessageListenerConcurrently() {  @Override  **public** ConsumeConcurrentlyStatus consumeMessage(  List<MessageExt> msgs, ConsumeConcurrentlyContext context) {  System.***out***.println(Thread.*currentThread*().getName()  + " Receive New Messages: " + msgs);  // 返回消费状态  // CONSUME\_SUCCESS 消费成功  // RECONSUME\_LATER 消费失败，需要稍后重新消费  **return** ConsumeConcurrentlyStatus.***CONSUME\_SUCCESS***;  }  });  // 调用start()方法启动consumer  consumer.start();  System.***out***.println("Consumer Started.");  }  } |

# 数据存储

## 数据库

### Mysql

#### 索引

关于[MySQL](http://lib.csdn.net/base/14)索引的好处，如果正确合理设计并且使用索引的MySQL是一辆兰博基尼的话，那么没有设计和使用索引的MySQL就是一个人力三轮车。对于没有索引的表，单表查询可能几十万数据就是瓶颈，而通常大型网站单日就可能会产生几十万甚至几百万的数据，没有索引查询会变的非常缓慢。还是以WordPress来说，其多个数据表都会对经常被查询的字段添加索引，比如wp\_comments表中针对5个字段设计了BTREE索引。

##### Mysql索引的概念

索引是一种特殊的文件(InnoDB数据表上的索引是表空间的一个组成部分)，它们包含着对数据表里所有记录的引用指针。更通俗的说，数据库索引好比是一本书前面的目录，能加快数据库的查询速度。上述SQL语句，在没有索引的情况下，数据库会遍历全部200条数据后选择符合条件的；而有了相应的索引之后，数据库会直接在索引中查找符合条件的选项。如果我们把SQL语句换成“SELECT \* FROM article WHERE id=2000000”，那么你是希望数据库按照顺序读取完200万行数据以后给你结果还是直接在索引中定位呢？上面的两个图片鲜明的用时对比已经给出了答案（注：一般数据库默认都会为主键生成索引）。

索引分为聚簇索引和非聚簇索引两种，聚簇索引是按照数据存放的物理位置为顺序的，而非聚簇索引就不一样了；聚簇索引能提高多行检索的速度，而非聚簇索引对于单行的检索很快。

**聚集索引：**

聚集索引基于数据行的键值，在表内排序和存储这些数据行。每个表只能有一个聚集索引，应为数据行本分只能按一个顺序存储。

在聚集索引中，表中各行的物理顺序与索引键值的逻辑（索引）顺序相同。聚集索引通常可加快UPDATE和DELETE操作的速度，因为这两个操作需要读取 大量的数据。创建或修改聚集索引可能要花很长时间，因为执行这两个操作时要在磁盘上对表的行进行重组。

**非聚集索引：**

因为一个表中只能有一个聚集索引，如果需要在表中建立多个索引，则可以创建为非聚集索引。表中的数据并不按照非聚集索引列的顺序存储，但非聚集索引的索引行中保存了非聚集键值和行定位器，可以快捷地根据非聚集键的值来定位记录的存储位置。

##### Mysql索引类型

**1、普通索引**

这是最基本的索引，它没有任何限制，比如上文中为title字段创建的索引就是一个普通索引，MyIASM中默认的BTREE类型的索引，也是我们大多数情况下用到的索引。

直接创建索引：CREATE INDEX index\_name ON table(column(length))

修改索引：ALTER TABLE table\_name ADD INDEX index\_name ON (column(length))

删除索引：DROP INDEX index\_name ON table

**2、唯一索引**

与普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值（注意和主键不同）。如果是组合索引，则列值的组合必须唯一，创建方法和普通索引类似。

创建唯一索引：CREATE UNIQUE INDEX indexName ON table(column(length))

修改唯一索引：ALTER TABLE table\_name ADD UNIQUE indexName ON (column(length))

**3、全文索引**

MySQL从3.23.23版开始支持全文索引和全文检索，FULLTEXT索引仅可用于 MyISAM 表；他们可以从CHAR、VARCHAR或TEXT列中作为CREATE TABLE语句的一部分被创建，或是随后使用ALTER TABLE 或CREATE INDEX被添加。////对于较大的数据集，将你的资料输入一个没有FULLTEXT索引的表中，然后创建索引，其速度比把资料输入现有FULLTEXT索引的速度更为快。不过切记对于大容量的数据表，生成全文索引是一个非常消耗时间非常消耗硬盘空间的做法。

创建全文索引：CREATE FULLTEXT INDEX index\_content ON article(content)

修改全文索引：ALTER TABLE article ADD FULLTEXT index\_content(content)

**4、单列索引、多列索引**

多个单列索引与单个多列索引的查询效果不同，因为执行查询时，MySQL只能使用一个索引，会从多个索引中选择一个限制最为严格的索引。

**5、组合索引（最左前缀）**

平时用的SQL查询语句一般都有比较多的限制条件，所以为了进一步榨取MySQL的效率，就要考虑建立组合索引。例如上表中针对title和time建立一个组合索引：ALTER TABLE article ADD INDEX index\_titme\_time (title(50),time(10))。建立这样的组合索引，其实是相当于分别建立了下面两组组合索引：

–title,time

–title

为什么没有time这样的组合索引呢？这是因为MySQL组合索引“最左前缀”的结果。简单的理解就是只从最左面的开始组合。

##### Mysql索引优化

上面都在说使用索引的好处，但过多的使用索引将会造成滥用。因此索引也会有它的缺点：虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在一个大表上创建了多种组合索引，索引文件的会膨胀很快。索引只是提高效率的一个因素，如果你的MySQL有[大数据](http://lib.csdn.net/base/20)量的表，就需要花时间研究建立最优秀的索引，或优化查询语句。下面是一些总结以及收藏的MySQL索引的注意事项和优化方法。

**1、何时使用聚集索引与非聚集索引**

|  |  |  |
| --- | --- | --- |
| 动作描述 | 使用聚集索引 | 使用非聚集索引 |
| 列经常被分组排序 | 使用 | 使用 |
| 返回某范围内的数据 | 使用 | 不使用 |
| 一个或极少不同值 | 不使用 | 不使用 |
| 小数目的不同值 | 使用 | 不使用 |
| 大数目的不同值 | 不使用 | 使用 |
| 频繁更新的列 | 不使用 | 使用 |
| 外键列 | 使用 | 使用 |
| 主键列 | 使用 | 使用 |
| 频繁修改索引列 | 不使用 | 使用 |

事实上，我们可以通过前面聚集索引和非聚集索引的定义的例子来理解上表。如：返回某范围内的数据一项。比如您的某个表有一个时间列，恰好您把聚合索引建立在了该列，这时您查询2004年1月1日至2004年10月1日之间的全部数据时，这个速度就将是很快的，因为您的这本字典正文是按日期进行排序的，聚类索引只需要找到要检索的所有数据中的开头和结尾数据即可；而不像非聚集索引，必须先查到目录中查到每一项数据对应的页码，然后再根据页码查到具体内容。其实这个具体用法我还不是很理解，只能等待后期的项目开发中慢慢学学了。

**2、索引不会包含有NULL值的列**

只要列中包含有NULL值都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。

**3、使用短索引**

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的列，如果在前10个或20个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

**4、索引列排序**

MySQL查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

**5、like语句操作**

一般情况下不鼓励使用like操作，如果非使用不可，如何使用也是一个问题。like “%aaa%” 不会使用索引而like “aaa%”可以使用索引。

**6、不要在列上进行计算**

例如：select \* from users where YEAR(adddate)<2007，将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成：select \* from users where adddate<’2007-01-01′。

最后总结一下，MySQL只对一下操作符才使用索引：<,<=,=,>,>=,between,in,以及某些时候的like(不以通配符%或\_开头的情形)。而理论上每张表里面最多可创建16个索引，不过除非是数据量真的很多，否则过多的使用索引也不是那么好玩的，比如我刚才针对text类型的字段创建索引的时候，系统差点就卡死了。

#### Explain查看执行计划

<https://www.cnblogs.com/butterfly100/archive/2018/01/15/8287569.html>

在日常工作中，我们会有时会开慢查询去记录一些执行时间比较久的SQL语句，找出这些SQL语句并不意味着完事了，些时我们常常用到explain这个命令来查看一个这些SQL语句的执行计划，查看该SQL语句有没有使用上了索引，有没有做全表扫描，这都可以通过explain命令来查看。所以我们深入了解MySQL的基于开销的优化器，还可以获得很多可能被优化器考虑到的访问策略的细节，以及当运行SQL语句时哪种策略预计会被优化器采用。（QEP：sql生成一个执行计划query Execution plan）

[复制代码](javascript:void(0);)

mysql> explain select \* from servers;

+----+-------------+---------+------+---------------+------+---------+------+------+-------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+---------+------+---------------+------+---------+------+------+-------+

| 1 | SIMPLE | servers | ALL | NULL | NULL | NULL | NULL | 1 | NULL |

+----+-------------+---------+------+---------------+------+---------+------+------+-------+

1 row in set (0.03 sec)

[复制代码](javascript:void(0);)

expain出来的信息有10列，分别是id、select\_type、table、type、possible\_keys、key、key\_len、ref、rows、Extra,下面对这些字段出现的可能进行解释：

##### id

**我的理解是SQL执行的顺序的标识,SQL从大到小的执行**

1. id相同时，执行顺序由上至下

2. 如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行

3.id如果相同，可以认为是一组，从上往下顺序执行；在所有组中，id值越大，优先级越高，越先执行

##### Select\_type

**表示查询中每个select子句的类型**

(1) SIMPLE(简单SELECT,不使用UNION或子查询等)

(2) PRIMARY(查询中若包含任何复杂的子部分,最外层的select被标记为PRIMARY)

(3) UNION(UNION中的第二个或后面的SELECT语句)

(4) DEPENDENT UNION(UNION中的第二个或后面的SELECT语句，取决于外面的查询)

(5) UNION RESULT(UNION的结果)

(6) SUBQUERY(子查询中的第一个SELECT)

(7) DEPENDENT SUBQUERY(子查询中的第一个SELECT，取决于外面的查询)

(8) DERIVED(派生表的SELECT, FROM子句的子查询)

(9) UNCACHEABLE SUBQUERY(一个子查询的结果不能被缓存，必须重新评估外链接的第一行)

##### Table

显示这一行的数据是关于哪张表的，有时不是真实的表名字,看到的是derivedx(x是个数字,我的理解是第几步执行的结果)

##### Type

表示MySQL在表中找到所需行的方式，又称“访问类型”。

常用的类型有：**ALL, index,  range, ref, eq\_ref, const, system, NULL（从左到右，性能从差到好）**

ALL：Full Table Scan， MySQL将遍历全表以找到匹配的行

index: Full Index Scan，index与ALL区别为index类型只遍历索引树

range:只检索给定范围的行，使用一个索引来选择行

ref: 表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值

eq\_ref: 类似ref，区别就在使用的索引是唯一索引，对于每个索引键值，表中只有一条记录匹配，简单来说，就是多表连接中使用primary key或者 unique key作为关联条件

const、system: 当MySQL对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于where列表中，MySQL就能将该查询转换为一个常量,system是const类型的特例，当查询的表只有一行的情况下，使用system

NULL: MySQL在优化过程中分解语句，执行时甚至不用访问表或索引，例如从一个索引列里选取最小值可以通过单独索引查找完成。

##### Possible\_keys

**指出MySQL能使用哪个索引在表中找到记录，查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用**

该列完全独立于EXPLAIN输出所示的表的次序。这意味着在possible\_keys中的某些键实际上不能按生成的表次序使用。  
如果该列是NULL，则没有相关的索引。在这种情况下，可以通过检查WHERE子句看是否它引用某些列或适合索引的列来提高你的查询性能。如果是这样，创造一个适当的索引并且再次用EXPLAIN检查查询

##### Key

**key列显示MySQL实际决定使用的键（索引）**

如果没有选择索引，键是NULL。要想强制MySQL使用或忽视possible\_keys列中的索引，在查询中使用FORCE INDEX、USE INDEX或者IGNORE INDEX。

##### Key\_len

**表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度（key\_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key\_len是根据表定义计算而得，不是通过表内检索出的）**

不损失精确性的情况下，长度越短越好

##### Ref

**表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值**

##### Rows

表示MySQL根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数

##### Extra

**该列包含MySQL解决查询的详细信息,有以下几种情况：**

Using where:列数据是从仅仅使用了索引中的信息而没有读取实际的行动的表返回的，这发生在对表的全部的请求列都是同一个索引的部分的时候，表示mysql服务器将在存储引擎检索行后再进行过滤

Using temporary：表示MySQL需要使用临时表来存储结果集，常见于排序和分组查询

Using filesort：MySQL中无法利用索引完成的排序操作称为“文件排序”

Using join buffer：改值强调了在获取连接条件时没有使用索引，并且需要连接缓冲区来存储中间结果。如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进能。

Impossible where：这个值强调了where语句会导致没有符合条件的行。

Select tables optimized away：这个值意味着仅通过使用索引，优化器可能仅从聚合函数结果中返回一行

##### 总结

**• EXPLAIN不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况**

**• EXPLAIN不考虑各种Cache**

**• EXPLAIN不能显示MySQL在执行查询时所作的优化工作**

**• 部分统计信息是估算的，并非精确值**

**• EXPALIN只能解释SELECT操作，其他操作要重写为SELECT后查看执行计划。**

### Oracle

## 缓存

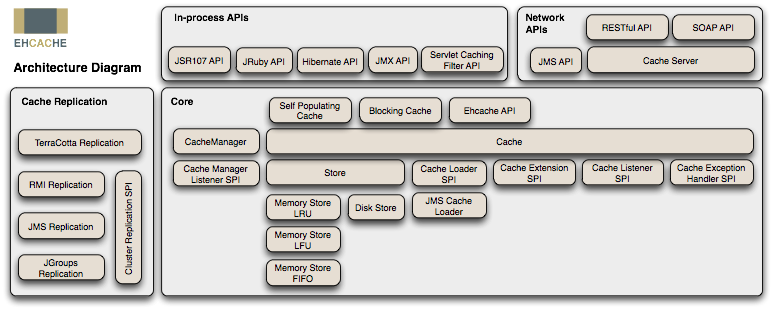
### Ehcache（Ehcache3）

在高并发应用中缓存就是核心机制。最近在研究Ehcache,发现这是一个更加灵活易用的缓存框架（相对于Redis、Memcache），Ehcache更加小巧轻便。而且都有持久化机制，不用担心JVM和服务器重启的数据丢失。我用四个字来形容：拎包入住。

著名的Hibernate的默认缓存策略就是用Ehcache，Liferay的缓存也是依赖Ehcache，可见其健壮性。与其黑盒的瞎眼使用，不如来研究下这后边的机制。

EhCache 是一个纯Java的进程内缓存框架，具有快速、精干等特点，是Hibernate中默认的CacheProvider。Ehcache是一种广泛使用的开 源Java分布式缓存。主要面向通用缓存,Java EE和轻量级容器。它具有内存和磁盘存储，缓存加载器,缓存扩展，缓存异常处理程序，一个gzip缓存servlet过滤器，支持REST和SOAP api等特点。

#### 架构



主要的特点：

1. 缓存数据有三级：内存、堆外缓存Off-Heap、Disk缓存，因此无需担心容量问题。还可以通过RMI、可插入API等方式进行分布式缓存。
2. 缓存数据会在虚拟机重启的过程中写入磁盘，持久化。
3. 具有缓存和缓存管理器的侦听接口。
4. 支持多缓存管理器实例，以及一个实例的多个缓存区域

#### 通用读写使用CacheManager

兼容3.0和2.0版

|  |
| --- |
| **package** org.xzh.springboot.ehcache;  **import** org.ehcache.Cache;  **import** org.ehcache.CacheManager;  **import** org.ehcache.config.builders.CacheConfigurationBuilder;  **import** org.ehcache.config.builders.CacheManagerBuilder;  **import** org.ehcache.config.builders.ResourcePoolsBuilder;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** MainTest {    **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(MainTest.**class**);  **public** **static** **void** main(String[] args) {  //创建cacheManager，在创建cacheManager的同时创建了cache  CacheManager cacheManager = CacheManagerBuilder  .*newCacheManagerBuilder*()  .withCache(  "preConfigured",  CacheConfigurationBuilder.*newCacheConfigurationBuilder*(  Long.**class**, String.**class**,  ResourcePoolsBuilder.*heap*(100)).build())  .build(**true**);  cacheManager.init();  //通过alias获取cache  Cache<Long, String> preConfigured = cacheManager.getCache(  "preConfigured", Long.**class**, String.**class**);  //通过cacheManager创建新的cache  Cache<Long, String> myCache = cacheManager.createCache(  "myCache",  CacheConfigurationBuilder.*newCacheConfigurationBuilder*(  Long.**class**, String.**class**,  ResourcePoolsBuilder.*heap*(100)).build());  //在cache加入缓存内容  myCache.put(1L, "da one!");  //获取缓存内容  String value = myCache.get(1L);    ***logger***.info(value);    //通过alias删除cache  cacheManager.removeCache("preConfigured");  cacheManager.close();  }  } |

#### 3.0新读写泛型方法UserMangedCache

|  |
| --- |
| **package** org.xzh.springboot.ehcache;  **import** org.ehcache.UserManagedCache;  **import** org.ehcache.config.builders.UserManagedCacheBuilder;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** UserManagedCacheMain {    **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(UserManagedCacheMain.**class**);  **public** **static** **void** main(String[] args) {  //创建UserManagedCache  UserManagedCache<Integer, String> userManagedCache = UserManagedCacheBuilder  .*newUserManagedCacheBuilder*(Integer.**class**, String.**class**).build();  //初始化UserManagedCache  userManagedCache.init();    //设置缓存  userManagedCache.put(1, "kkk");  ***logger***.info(userManagedCache.get(1));    //关闭userManagedCache  userManagedCache.close();  }  } |

#### 三级缓存策略

之前的版本是2级缓存，内存和磁盘，新版增加了更灵活的一级，堆外缓存（off-heap），这既是独立的进程缓存，还是JVM堆外的系统缓存，可以想象一下，JVM堆是非常宝贵的，如果占用过大会带来GC性能问题，堆外缓存很好的解决了这个问题，现在服务器内存越来越大，而磁盘缓存的io性能又比较低，off-heap缓存就是折中的方案，既保证了高速性能，又可以有一定的容量。off-heap缓存性能的占用主要是序列化、反序列化的过程。一旦对象被序列化，在返回Java堆的时候必需反序列化才可以使用。这是一笔性能开销。但off-heap还是要比本地磁盘、网络存储、RDBMS数据库IO等记录数据的系统要快非常多。还应指出的是，序列化/反序列化的性能开销远没有很多用户想象的那么大。off-heap已经针对字节缓冲区做了优化，本身也包含一些优化机制，可以对使用标准Java序列化的对象进行优化，能使复杂Java对象的性能提升两倍，使byte数组的性能提升四倍。

个人是这样理解的：

1. 常被查询、最重要、数据量较小的数据存放在堆缓存，不用担心JVM的重启，有持久化机制；
2. 常被查询、数据量中等的数据存放在堆外缓存，几个G就好了，不用担心服务器的重启，有持久化机制；
3. 不常用、大量的数据、但又不想占用数据库IO的数据，放在Disk缓存，容量自便；

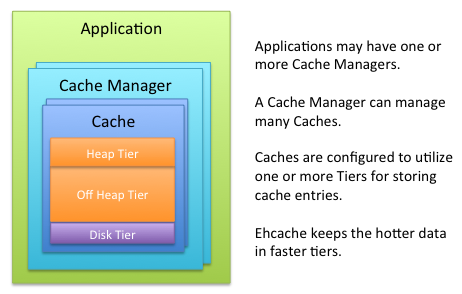
　　理解了三级缓存机制，现在还有一个黑盒问题有待了解，即EHcache的持久化策略算法，Redis的持久化策略是2种，即RDB快照和AOF追加日志，让用户在性能和完整性之间来自由选择，这非常灵活。目前还没查到EHcache的持久化策略，本好奇猫有点沮丧。

|  |
| --- |
| **package** org.xzh.springboot.ehcache;  **import** java.io.File;  **import** org.ehcache.Cache;  **import** org.ehcache.PersistentCacheManager;  **import** org.ehcache.config.builders.CacheConfigurationBuilder;  **import** org.ehcache.config.builders.CacheManagerBuilder;  **import** org.ehcache.config.builders.ResourcePoolsBuilder;  **import** org.ehcache.config.units.EntryUnit;  **import** org.ehcache.config.units.MemoryUnit;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** PersistentCacheManagerMain {    **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(PersistentCacheManagerMain.**class**);  **public** **static** **void** main(String[] args) {  PersistentCacheManager persistentCacheManager = CacheManagerBuilder  .*newCacheManagerBuilder*()  .with(CacheManagerBuilder.*persistence*("E:" + File.***separator*** + "myData"))  .withCache("threeTieredCache",  CacheConfigurationBuilder.*newCacheConfigurationBuilder*(Integer.**class**, String.**class**,  ResourcePoolsBuilder.*newResourcePoolsBuilder*()  .heap(10, EntryUnit.***ENTRIES***) //堆  .offheap(1, MemoryUnit.***MB***) //堆外  .disk(20, MemoryUnit.***GB***) //磁盘  )  ).build(**true**);    Cache<Integer, String> threeTieredCache = persistentCacheManager  .getCache("threeTieredCache", Integer.**class**, String.**class**);    //写  **for**(**int** i=0; i<20000; i++){  threeTieredCache.put(i, "$"+i);  }    //读  **for**(**int** i=0; i<20000; i++){  ***logger***.info(threeTieredCache.get(i));  }    persistentCacheManager.close();  }  } |

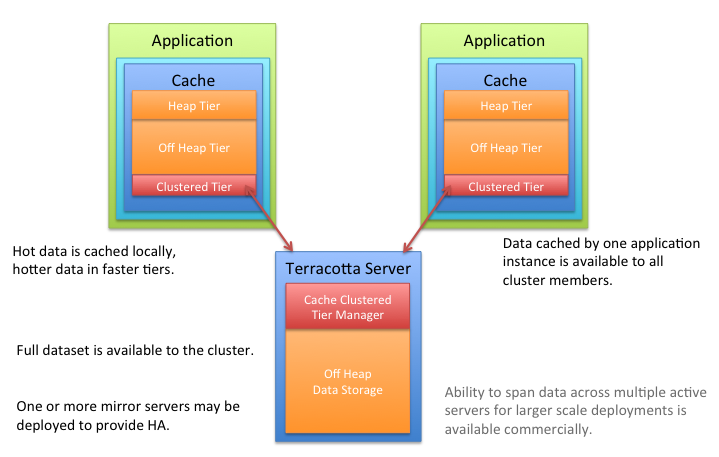
#### 集群

不用集群的多级缓存

应用程序直接访问一个或者多个Cache Manager，而一个Cache Manager管理Caches集合。Caches集合可以多级存储。



使用集群



#### 一些更高级用法事务缓存

对读写原子性有要求的必读：

http://www.ehcache.org/documentation/3.3/xa.html线程池的使用

用于异步操作，这将大幅度提升效率：

http://www.ehcache.org/documentation/3.3/thread-pools.html

#### Ehcache与Spring结合

### Memcached

Memcached是一个自由开源的，高性能，分布式内存对象缓存系统。

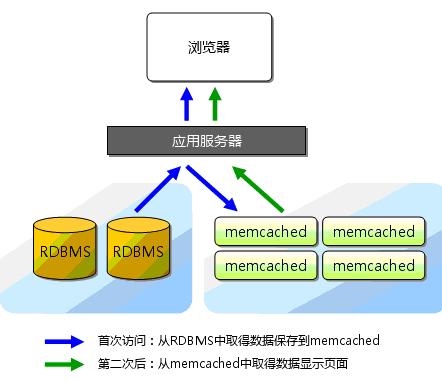
Memcached是以LiveJournal旗下Danga Interactive公司的Brad Fitzpatric为首开发的一款软件。现在已成为mixi、hatena、Facebook、Vox、LiveJournal等众多服务中提高Web应用扩展性的重要因素。

Memcached是一种基于内存的key-value存储，用来存储小块的任意数据（字符串、对象）。这些数据可以是数据库调用、API调用或者是页面渲染的结果。

Memcached简洁而强大。它的简洁设计便于快速开发，减轻开发难度，解决了大数据量缓存的很多问题。它的API兼容大部分流行的开发语言。

本质上，它是一个简洁的key-value存储系统。

一般的使用目的是，通过缓存数据库查询结果，减少数据库访问次数，以提高动态Web应用的速度、提高可扩展性。



#### 安装memcached

http://www.runoob.com/memcached/memcached-install.html

#### memcached存储命令

##### set

Memcached set 命令用于将 **value(数据值)** 存储在指定的 **key(键)** 中。

如果set的key已经存在，该命令可以更新该key所对应的原来的数据，也就是实现更新的作用。语法：

set 命令的基本语法格式如下：

set key flags exptime bytes [noreply]

value

参数说明如下：

* **key：**键值 key-value 结构中的 key，用于查找缓存值。
* **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息 。
* **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
* **bytes**：在缓存中存储的字节数
* **noreply（可选）**： 该参数告知服务器不需要返回数据
* **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

以下实例中我们设置：

* key → runoob
* flag → 0
* exptime → 900 (以秒为单位)
* bytes → 9 (数据存储的字节数)
* value → memcached

set runoob 0 900 9

memcached

STORED

get runoob

VALUE runoob 0 9

memcached

END

输出

如果数据设置成功，则输出：

STORED

输出信息说明：

* **STORED**：保存成功后输出。
* **ERROR**：在保存失败后输出。

##### Add

Memcached add 命令用于将 **value(数据值)** 存储在指定的 **key(键)** 中。

如果 add 的 key 已经存在，则不会更新数据(过期的 key 会更新)，之前的值将仍然保持相同，并且您将获得响应 **NOT\_STORED**。

语法：

add 命令的基本语法格式如下：

add key flags exptime bytes [noreply]

value

参数说明如下：

* **key：**键值 key-value 结构中的 key，用于查找缓存值。
* **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息 。
* **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
* **bytes**：在缓存中存储的字节数
* **noreply（可选）**： 该参数告知服务器不需要返回数据
* **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

以下实例中我们设置：

* key → new\_key
* flag → 0
* exptime → 900 (以秒为单位)
* bytes → 10 (数据存储的字节数)
* value → data\_value

add new\_key 0 900 10

data\_value

STORED

get new\_key

VALUE new\_key 0 10

data\_value

END

输出

如果数据添加成功，则输出：

STORED

输出信息说明：

* **STORED**：保存成功后输出。
* **NOT\_STORED**：在保存失败后输出。

##### Replace

Memcached replace 命令用于替换已存在的 **key(键)** 的 **value(数据值)**。

如果 key 不存在，则替换失败，并且您将获得响应 **NOT\_STORED**。

语法：

replace 命令的基本语法格式如下：

replace key flags exptime bytes [noreply]

value

参数说明如下：

* **key：**键值 key-value 结构中的 key，用于查找缓存值。
* **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息 。
* **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
* **bytes**：在缓存中存储的字节数
* **noreply（可选）**： 该参数告知服务器不需要返回数据
* **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

以下实例中我们设置：

* key → mykey
* flag → 0
* exptime → 900 (以秒为单位)
* bytes → 10 (数据存储的字节数)
* value → data\_value

以下实例中我们使用的键位 'mykey' 并存储对应的值 data\_value。执行后我们替换相同的 key 的值为 'some\_other\_value'。

add mykey 0 900 10

data\_value

STORED

get mykey

VALUE mykey 0 10

data\_value

END

replace mykey 0 900 16

some\_other\_value

get mykey

VALUE mykey 0 16

some\_other\_value

END

输出

如果数据添加成功，则输出：

STORED

输出信息说明：

* **STORED**：保存成功后输出。
* **NOT\_STORED**：执行替换失败后输出。

##### Append

Memcached append 命令用于向已存在 **key(键)** 的 **value(数据值)** 后面追加数据 。

语法：

append 命令的基本语法格式如下：

append key flags exptime bytes [noreply]

value

参数说明如下：

* **key：**键值 key-value 结构中的 key，用于查找缓存值。
* **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息 。
* **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
* **bytes**：在缓存中存储的字节数
* **noreply（可选）**： 该参数告知服务器不需要返回数据
* **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

实例如下：

* 首先我们在 Memcached 中存储一个键 runoob，其值为 memcached。
* 然后，我们使用 get 命令检索该值。
* 然后，我们使用 **append** 命令在键为 runoob 的值后面追加 "redis"。
* 最后，我们再使用 get 命令检索该值。

set runoob 0 900 9

memcached

STORED

get runoob

VALUE runoob 0 9

memcached

END

append runoob 0 900 5

redis

STORED

get runoob

VALUE runoob 0 14

memcachedredis

END

输出

如果数据添加成功，则输出：

STORED

输出信息说明：

* **STORED**：保存成功后输出。
* **NOT\_STORED**：该键在 Memcached 上不存在。
* **CLIENT\_ERROR**：执行错误。

##### Prepend

Memcached prepend 命令用于向已存在 **key(键)** 的 **value(数据值)** 前面追加数据 。

语法：

prepend 命令的基本语法格式如下：

prepend key flags exptime bytes [noreply]

value

参数说明如下：

* **key：**键值 key-value 结构中的 key，用于查找缓存值。
* **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息 。
* **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
* **bytes**：在缓存中存储的字节数
* **noreply（可选）**： 该参数告知服务器不需要返回数据
* **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

实例如下：

* 首先我们在 Memcached 中存储一个键 runoob，其值为 memcached。
* 然后，我们使用 get 命令检索该值。
* 然后，我们使用 **prepend** 命令在键为 runoob 的值前面追加 "redis"。
* 最后，我们再使用 get 命令检索该值。

set runoob 0 900 9

memcached

STORED

get runoob

VALUE runoob 0 9

memcached

END

prepend runoob 0 900 5

redis

STORED

get runoob

VALUE runoob 0 14

redismemcached

END

输出

如果数据添加成功，则输出：

STORED

输出信息说明：

* **STORED**：保存成功后输出。
* **NOT\_STORED**：该键在 Memcached 上不存在。
* **CLIENT\_ERROR**：执行错误。

#### memcached查找命令

##### get

Memcached get 命令获取存储在 **key(键)** 中的 **value(数据值)** ，如果 key 不存在，则返回空。

语法：

get 命令的基本语法格式如下：

get key

多个 key 使用空格隔开，如下:

get key1 key2 key3

参数说明如下：

* **key：**键值 key-value 结构中的 key，用于查找缓存值。

实例

在以下实例中，我们使用 runoob 作为 key，过期时间设置为 900 秒。

set runoob 0 900 9

memcached

STORED

get runoob

VALUE runoob 0 9

memcached

END

##### Gets

Memcached gets 命令获取带有 CAS 令牌存 的 **value(数据值)** ，如果 key 不存在，则返回空。

语法：

gets 命令的基本语法格式如下：

gets key

多个 key 使用空格隔开，如下:

gets key1 key2 key3

参数说明如下：

* **key：**键值 key-value 结构中的 key，用于查找缓存值。

实例

在以下实例中，我们使用 runoob 作为 key，过期时间设置为 900 秒。

set runoob 0 900 9

memcached

STORED

gets runoob

VALUE runoob 0 9 1

memcached

END

在 使用 gets 命令的输出结果中，在最后一列的数字 1 代表了 key 为 runoob 的 CAS 令牌。

##### Delete

Memcached delete 命令用于删除已存在的 key(键)。

语法：

delete 命令的基本语法格式如下：

delete key [noreply]

参数说明如下：

* **key：**键值 key-value 结构中的 key，用于查找缓存值。
* **noreply（可选）**： 该参数告知服务器不需要返回数据

实例

在以下实例中，我们使用 runoob 作为 key，过期时间设置为 900 秒。之后我们使用 delete 命令删除该 key。

set runoob 0 900 9

memcached

STORED

get runoob

VALUE runoob 0 9

memcached

END

delete runoob

DELETED

get runoob

END

delete runoob

NOT\_FOUND

输出

输出信息说明：

* **DELETED**：删除成功。
* **ERROR**：语法错误或删除失败。
* **NOT\_FOUND**：key 不存在。

##### Incr与decr

Memcached incr 与 decr 命令用于对已存在的 key(键) 的数字值进行自增或自减操作。

incr 与 decr 命令操作的数据必须是十进制的32位无符号整数。

如果 key 不存在返回 **NOT\_FOUND**，如果键的值不为数字，则返回 **CLIENT\_ERROR**，其他错误返回 **ERROR**。

incr 命令

语法：

incr 命令的基本语法格式如下：

incr key increment\_value

参数说明如下：

* **key：**键值 key-value 结构中的 key，用于查找缓存值。
* **increment\_value**： 增加的数值。

实例

在以下实例中，我们使用 visitors 作为 key，初始值为 10，之后进行加 5 操作。

set visitors 0 900 2

10

STORED

get visitors

VALUE visitors 0 2

10

END

incr visitors 5

15

get visitors

VALUE visitors 0 2

15

END

输出

输出信息说明：

* **NOT\_FOUND**：key 不存在。
* **CLIENT\_ERROR**：自增值不是对象。
* **ERROR**其他错误，如语法错误等。

decr 命令

decr 命令的基本语法格式如下：

decr key decrement\_value

参数说明如下：

* **key：**键值 key-value 结构中的 key，用于查找缓存值。
* **decrement\_value**： 减少的数值。

实例

set visitors 0 900 2

10

STORED

get visitors

VALUE visitors 0 2

10

END

decr visitors 5

5

get visitors

VALUE visitors 0 1

5

END

在以下实例中，我们使用 visitors 作为 key，初始值为 10，之后进行减 5 操作。

输出

输出信息说明：

* **NOT\_FOUND**：key 不存在。
* **CLIENT\_ERROR**：自增值不是对象。
* **ERROR**其他错误，如语法错误等。

#### Memcached统计命令

##### Stats

#### Java用法

### redis

#### 安装redis

#### 简介

redis是一种高级的key:value存储系统，其中value支持五种数据类型：

1.字符串（strings）  
2.字符串列表（lists）  
3.字符串集合（sets）  
4.有序字符串集合（sorted sets）  
5.哈希（hashes）

而关于key，有几个点要提醒大家：

1.key不要太长，尽量不要超过1024字节，这不仅消耗内存，而且会降低查找的效率；  
2.key也不要太短，太短的话，key的可读性会降低；  
3.在一个项目中，key最好使用统一的命名模式，例如user:10000:passwd。

#### 字符串（Strings）数据结构

有人说，如果只使用redis中的字符串类型，且不使用redis的持久化功能，那么，redis就和memcache非常非常的像了。这说明strings类型是一个很基础的数据类型，也是任何存储系统都必备的数据类型。

我们来看一个最简单的例子：

复制代码代码如下:

set mystr "hello world!" //设置字符串类型  
get mystr //读取字符串类型

字符串类型的用法就是这么简单，因为是二进制安全的，所以你完全可以把一个图片文件的内容作为字符串来存储。

另外，我们还可以通过字符串类型进行数值操作：

复制代码代码如下:

127.0.0.1:6379> set mynum "2"  
OK  
127.0.0.1:6379> get mynum  
"2"  
127.0.0.1:6379> incr mynum  
(integer) 3  
127.0.0.1:6379> get mynum  
"3"

看，在遇到数值操作时，redis会将字符串类型转换成数值。

由于INCR等指令本身就具有原子操作的特性，所以我们完全可以利用redis的INCR、INCRBY、DECR、DECRBY等指令来实现原子计数的效果，假如，在某种场景下有3个客户端同时读取了mynum的值（值为2），然后对其同时进行了加1的操作，那么，最后mynum的值一定是5。不少网站都利用redis的这个特性来实现业务上的统计计数需求。

#### 列表（lists）数据结构

redis的另一个重要的数据结构叫做lists，翻译成中文叫做“列表”。

首先要明确一点，redis中的lists在底层实现上并不是数组，而是链表，也就是说对于一个具有上百万个元素的lists来说，在头部和尾部插入一个新元素，其时间复杂度是常数级别的，比如用LPUSH在10个元素的lists头部插入新元素，和在上千万元素的lists头部插入新元素的速度应该是相同的。

虽然lists有这样的优势，但同样有其弊端，那就是，链表型lists的元素定位会比较慢，而数组型lists的元素定位就会快得多。

lists的常用操作包括LPUSH、RPUSH、LRANGE等。我们可以用LPUSH在lists的左侧插入一个新元素，用RPUSH在lists的右侧插入一个新元素，用LRANGE命令从lists中指定一个范围来提取元素。我们来看几个例子：

复制代码代码如下:

//新建一个list叫做mylist，并在列表头部插入元素"1"  
127.0.0.1:6379> lpush mylist "1"   
//返回当前mylist中的元素个数  
(integer) 1   
//在mylist右侧插入元素"2"  
127.0.0.1:6379> rpush mylist "2"   
(integer) 2  
//在mylist左侧插入元素"0"  
127.0.0.1:6379> lpush mylist "0"   
(integer) 3  
//列出mylist中从编号0到编号1的元素  
127.0.0.1:6379> lrange mylist 0 1   
1) "0"  
2) "1"  
//列出mylist中从编号0到倒数第一个元素  
127.0.0.1:6379> lrange mylist 0 -1   
1) "0"  
2) "1"  
3) "2"

lists的应用相当广泛，随便举几个例子：

1.我们可以利用lists来实现一个消息队列，而且可以确保先后顺序，不必像MySQL那样还需要通过ORDER BY来进行排序。  
2.利用LRANGE还可以很方便的实现分页的功能。  
3.在博客系统中，每片博文的评论也可以存入一个单独的list中。

#### 集合（sets）数据结构

redis的集合，是一种无序的集合，集合中的元素没有先后顺序。

集合相关的操作也很丰富，如添加新元素、删除已有元素、取交集、取并集、取差集等。我们来看例子：

复制代码代码如下:

//向集合myset中加入一个新元素"one"  
127.0.0.1:6379> sadd myset "one"   
(integer) 1  
127.0.0.1:6379> sadd myset "two"  
(integer) 1  
//列出集合myset中的所有元素  
127.0.0.1:6379> smembers myset   
1) "one"  
2) "two"  
//判断元素1是否在集合myset中，返回1表示存在  
127.0.0.1:6379> sismember myset "one"   
(integer) 1  
//判断元素3是否在集合myset中，返回0表示不存在  
127.0.0.1:6379> sismember myset "three"   
(integer) 0  
//新建一个新的集合yourset  
127.0.0.1:6379> sadd yourset "1"   
(integer) 1  
127.0.0.1:6379> sadd yourset "2"  
(integer) 1  
127.0.0.1:6379> smembers yourset  
1) "1"  
2) "2"  
//对两个集合求并集  
127.0.0.1:6379> sunion myset yourset   
1) "1"  
2) "one"  
3) "2"  
4) "two"

对于集合的使用，也有一些常见的方式，比如，QQ有一个社交功能叫做“好友标签”，大家可以给你的好友贴标签，比如“大美女”、“土豪”、“欧巴”等等，这时就可以使用redis的集合来实现，把每一个用户的标签都存储在一个集合之中。

#### 有序集合（sorted sets）数据结构

redis不但提供了无需集合（sets），还很体贴的提供了有序集合（sorted sets）。有序集合中的每个元素都关联一个序号（score），这便是排序的依据。

很多时候，我们都将redis中的有序集合叫做zsets，这是因为在redis中，有序集合相关的操作指令都是以z开头的，比如zrange、zadd、zrevrange、zrangebyscore等等

老规矩，我们来看几个生动的例子：  
//新增一个有序集合myzset，并加入一个元素baidu.com，给它赋予的序号是1：

复制代码代码如下:

127.0.0.1:6379> zadd myzset 1 baidu.com   
(integer) 1  
//向myzset中新增一个元素360.com，赋予它的序号是3  
127.0.0.1:6379> zadd myzset 3 360.com   
(integer) 1  
//向myzset中新增一个元素google.com，赋予它的序号是2  
127.0.0.1:6379> zadd myzset 2 google.com   
(integer) 1  
//列出myzset的所有元素，同时列出其序号，可以看出myzset已经是有序的了。  
127.0.0.1:6379> zrange myzset 0 -1 with scores   
1) "baidu.com"  
2) "1"  
3) "google.com"  
4) "2"  
5) "360.com"  
6) "3"  
//只列出myzset的元素  
127.0.0.1:6379> zrange myzset 0 -1   
1) "baidu.com"  
2) "google.com"  
3) "360.com"

#### 哈希（hashes）数据结构

最后要给大家介绍的是hashes，即哈希。哈希是从redis-2.0.0版本之后才有的数据结构。

hashes存的是字符串和字符串值之间的映射，比如一个用户要存储其全名、姓氏、年龄等等，就很适合使用哈希。

我们来看一个例子：

复制代码代码如下:

//建立哈希，并赋值  
127.0.0.1:6379> HMSET user:001 username antirez password P1pp0 age 34   
OK  
//列出哈希的内容  
127.0.0.1:6379> HGETALL user:001   
1) "username"  
2) "antirez"  
3) "password"  
4) "P1pp0"  
5) "age"  
6) "34"  
//更改哈希中的某一个值  
127.0.0.1:6379> HSET user:001 password 12345   
(integer) 0  
//再次列出哈希的内容  
127.0.0.1:6379> HGETALL user:001   
1) "username"  
2) "antirez"  
3) "password"  
4) "12345"  
5) "age"  
6) "34"

#### Reids持久化

redis提供了两种持久化的方式，分别是RDB（Redis DataBase）和AOF（Append Only File）。

RDB，简而言之，就是在不同的时间点，将redis存储的数据生成快照并存储到磁盘等介质上；

AOF，则是换了一个角度来实现持久化，那就是将redis执行过的所有写指令记录下来，在下次redis重新启动时，只要把这些写指令从前到后再重复执行一遍，就可以实现数据恢复了。

其实RDB和AOF两种方式也可以同时使用，在这种情况下，如果redis重启的话，则会优先采用AOF方式来进行数据恢复，这是因为AOF方式的数据恢复完整度更高。

如果你没有数据持久化的需求，也完全可以关闭RDB和AOF方式，这样的话，redis将变成一个纯内存数据库，就像memcache一样。

**【聊聊redis持久化 – RDB】**

RDB方式，是将redis某一时刻的数据持久化到磁盘中，是一种快照式的持久化方法。

redis在进行数据持久化的过程中，会先将数据写入到一个临时文件中，待持久化过程都结束了，才会用这个临时文件替换上次持久化好的文件。正是这种特性，让我们可以随时来进行备份，因为快照文件总是完整可用的。

对于RDB方式，redis会单独创建（fork）一个子进程来进行持久化，而主进程是不会进行任何IO操作的，这样就确保了redis极高的性能。

如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。

虽然RDB有不少优点，但它的缺点也是不容忽视的。如果你对数据的完整性非常敏感，那么RDB方式就不太适合你，因为即使你每5分钟都持久化一次，当redis故障时，仍然会有近5分钟的数据丢失。所以，redis还提供了另一种持久化方式，那就是AOF。

**【聊聊redis持久化 – AOF】**

AOF，英文是Append Only File，即只允许追加不允许改写的文件。

如前面介绍的，AOF方式是将执行过的写指令记录下来，在数据恢复时按照从前到后的顺序再将指令都执行一遍，就这么简单。

我们通过配置redis.conf中的appendonly yes就可以打开AOF功能。如果有写操作（如SET等），redis就会被追加到AOF文件的末尾。

默认的AOF持久化策略是每秒钟fsync一次（fsync是指把缓存中的写指令记录到磁盘中），因为在这种情况下，redis仍然可以保持很好的处理性能，即使redis故障，也只会丢失最近1秒钟的数据。

如果在追加日志时，恰好遇到磁盘空间满、inode满或断电等情况导致日志写入不完整，也没有关系，redis提供了redis-check-aof工具，可以用来进行日志修复。

因为采用了追加方式，如果不做任何处理的话，AOF文件会变得越来越大，为此，redis提供了AOF文件重写（rewrite）机制，即当AOF文件的大小超过所设定的阈值时，redis就会启动AOF文件的内容压缩，只保留可以恢复数据的最小指令集。举个例子或许更形象，假如我们调用了100次INCR指令，在AOF文件中就要存储100条指令，但这明显是很低效的，完全可以把这100条指令合并成一条SET指令，这就是重写机制的原理。

在进行AOF重写时，仍然是采用先写临时文件，全部完成后再替换的流程，所以断电、磁盘满等问题都不会影响AOF文件的可用性，这点大家可以放心。

AOF方式的另一个好处，我们通过一个“场景再现”来说明。某同学在操作redis时，不小心执行了FLUSHALL，导致redis内存中的数据全部被清空了，这是很悲剧的事情。不过这也不是世界末日，只要redis配置了AOF持久化方式，且AOF文件还没有被重写（rewrite），我们就可以用最快的速度暂停redis并编辑AOF文件，将最后一行的FLUSHALL命令删除，然后重启redis，就可以恢复redis的所有数据到FLUSHALL之前的状态了。是不是很神奇，这就是AOF持久化方式的好处之一。但是如果AOF文件已经被重写了，那就无法通过这种方法来恢复数据了。

虽然优点多多，但AOF方式也同样存在缺陷，比如在同样数据规模的情况下，AOF文件要比RDB文件的体积大。而且，AOF方式的恢复速度也要慢于RDB方式。

如果你直接执行BGREWRITEAOF命令，那么redis会生成一个全新的AOF文件，其中便包括了可以恢复现有数据的最少的命令集。

如果运气比较差，AOF文件出现了被写坏的情况，也不必过分担忧，redis并不会贸然加载这个有问题的AOF文件，而是报错退出。这时可以通过以下步骤来修复出错的文件：

1.备份被写坏的AOF文件  
2.运行redis-check-aof –fix进行修复  
3.用diff -u来看下两个文件的差异，确认问题点  
4.重启redis，加载修复后的AOF文件

**【聊聊redis持久化 – AOF重写】**

AOF重写的内部运行原理，我们有必要了解一下。

在重写即将开始之际，redis会创建（fork）一个“重写子进程”，这个子进程会首先读取现有的AOF文件，并将其包含的指令进行分析压缩并写入到一个临时文件中。

与此同时，主工作进程会将新接收到的写指令一边累积到内存缓冲区中，一边继续写入到原有的AOF文件中，这样做是保证原有的AOF文件的可用性，避免在重写过程中出现意外。

当“重写子进程”完成重写工作后，它会给父进程发一个信号，父进程收到信号后就会将内存中缓存的写指令追加到新AOF文件中。

当追加结束后，redis就会用新AOF文件来代替旧AOF文件，之后再有新的写指令，就都会追加到新的AOF文件中了。

**【聊聊redis持久化 – 如何选择RDB和AOF】**

对于我们应该选择RDB还是AOF，官方的建议是两个同时使用。这样可以提供更可靠的持久化方案。

#### Redis主从模式

像MySQL一样，redis是支持主从同步的，而且也支持一主多从以及多级从结构。

主从结构，一是为了纯粹的冗余备份，二是为了提升读性能，比如很消耗性能的SORT就可以由从服务器来承担。

redis的主从同步是异步进行的，这意味着主从同步不会影响主逻辑，也不会降低redis的处理性能。

主从架构中，可以考虑关闭主服务器的数据持久化功能，只让从服务器进行持久化，这样可以提高主服务器的处理性能。

在主从架构中，从服务器通常被设置为只读模式，这样可以避免从服务器的数据被误修改。但是从服务器仍然可以接受CONFIG等指令，所以还是不应该将从服务器直接暴露到不安全的网络环境中。如果必须如此，那可以考虑给重要指令进行重命名，来避免命令被外人误执行。

**【聊聊主从 – 同步原理】**

从服务器会向主服务器发出SYNC指令，当主服务器接到此命令后，就会调用BGSAVE指令来创建一个子进程专门进行数据持久化工作，也就是将主服务器的数据写入RDB文件中。在数据持久化期间，主服务器将执行的写指令都缓存在内存中。

在BGSAVE指令执行完成后，主服务器会将持久化好的RDB文件发送给从服务器，从服务器接到此文件后会将其存储到磁盘上，然后再将其读取到内存中。这个动作完成后，主服务器会将这段时间缓存的写指令再以redis协议的格式发送给从服务器。

另外，要说的一点是，即使有多个从服务器同时发来SYNC指令，主服务器也只会执行一次BGSAVE，然后把持久化好的RDB文件发给多个下游。在redis2.8版本之前，如果从服务器与主服务器因某些原因断开连接的话，都会进行一次主从之间的全量的数据同步；而在2.8版本之后，redis支持了效率更高的增量同步策略，这大大降低了连接断开的恢复成本。

主服务器会在内存中维护一个缓冲区，缓冲区中存储着将要发给从服务器的内容。从服务器在与主服务器出现网络瞬断之后，从服务器会尝试再次与主服务器连接，一旦连接成功，从服务器就会把“希望同步的主服务器ID”和“希望请求的数据的偏移位置（replication offset）”发送出去。主服务器接收到这样的同步请求后，首先会验证主服务器ID是否和自己的ID匹配，其次会检查“请求的偏移位置”是否存在于自己的缓冲区中，如果两者都满足的话，主服务器就会向从服务器发送增量内容。

增量同步功能，需要服务器端支持全新的PSYNC指令。这个指令，只有在redis-2.8之后才具有。

#### Redis事物处理

众所周知，事务是指“一个完整的动作，要么全部执行，要么什么也没有做”。

在聊redis事务处理之前，要先和大家介绍四个redis指令，即MULTI、EXEC、DISCARD、WATCH。这四个指令构成了redis事务处理的基础。

1.MULTI用来组装一个事务；  
2.EXEC用来执行一个事务；  
3.DISCARD用来取消一个事务；  
4.WATCH用来监视一些key，一旦这些key在事务执行之前被改变，则取消事务的执行。

纸上得来终觉浅，我们来看一个MULTI和EXEC的例子：

复制代码代码如下:

redis> MULTI //标记事务开始  
OK  
redis> INCR user\_id //多条命令按顺序入队  
QUEUED  
redis> INCR user\_id  
QUEUED  
redis> INCR user\_id  
QUEUED  
redis> PING  
QUEUED  
redis> EXEC //执行  
1) (integer) 1  
2) (integer) 2  
3) (integer) 3  
4) PONG

在上面的例子中，我们看到了QUEUED的字样，这表示我们在用MULTI组装事务时，每一个命令都会进入到内存队列中缓存起来，如果出现QUEUED则表示我们这个命令成功插入了缓存队列，在将来执行EXEC时，这些被QUEUED的命令都会被组装成一个事务来执行。

对于事务的执行来说，如果redis开启了AOF持久化的话，那么一旦事务被成功执行，事务中的命令就会通过write命令一次性写到磁盘中去，如果在向磁盘中写的过程中恰好出现断电、硬件故障等问题，那么就可能出现只有部分命令进行了AOF持久化，这时AOF文件就会出现不完整的情况，这时，我们可以使用redis-check-aof工具来修复这一问题，这个工具会将AOF文件中不完整的信息移除，确保AOF文件完整可用。

有关事务，大家经常会遇到的是两类错误：

1.调用EXEC之前的错误  
2.调用EXEC之后的错误

“调用EXEC之前的错误”，有可能是由于语法有误导致的，也可能时由于内存不足导致的。只要出现某个命令无法成功写入缓冲队列的情况，redis都会进行记录，在客户端调用EXEC时，redis会拒绝执行这一事务。（这时2.6.5版本之后的策略。在2.6.5之前的版本中，redis会忽略那些入队失败的命令，只执行那些入队成功的命令）。我们来看一个这样的例子：

复制代码代码如下:

127.0.0.1:6379> multi  
OK  
127.0.0.1:6379> haha //一个明显错误的指令  
(error) ERR unknown command 'haha'  
127.0.0.1:6379> ping  
QUEUED  
127.0.0.1:6379> exec  
//redis无情的拒绝了事务的执行，原因是“之前出现了错误”  
(error) EXECABORT Transaction discarded because of previous errors.

而对于“调用EXEC之后的错误”，redis则采取了完全不同的策略，即redis不会理睬这些错误，而是继续向下执行事务中的其他命令。这是因为，对于应用层面的错误，并不是redis自身需要考虑和处理的问题，所以一个事务中如果某一条命令执行失败，并不会影响接下来的其他命令的执行。我们也来看一个例子：

复制代码代码如下:

127.0.0.1:6379> multi  
OK  
127.0.0.1:6379> set age 23  
QUEUED  
//age不是集合，所以如下是一条明显错误的指令  
127.0.0.1:6379> sadd age 15   
QUEUED  
127.0.0.1:6379> set age 29  
QUEUED  
127.0.0.1:6379> exec //执行事务时，redis不会理睬第2条指令执行错误  
1) OK  
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value  
3) OK  
127.0.0.1:6379> get age  
"29" //可以看出第3条指令被成功执行了

好了，我们来说说最后一个指令“WATCH”，这是一个很好用的指令，它可以帮我们实现类似于“乐观锁”的效果，即CAS（check and set）。

WATCH本身的作用是“监视key是否被改动过”，而且支持同时监视多个key，只要还没真正触发事务，WATCH都会尽职尽责的监视，一旦发现某个key被修改了，在执行EXEC时就会返回nil，表示事务无法触发。

复制代码代码如下:

127.0.0.1:6379> set age 23  
OK  
127.0.0.1:6379> watch age //开始监视age  
OK  
127.0.0.1:6379> set age 24 //在EXEC之前，age的值被修改了  
OK  
127.0.0.1:6379> multi  
OK  
127.0.0.1:6379> set age 25  
QUEUED  
127.0.0.1:6379> get age  
QUEUED  
127.0.0.1:6379> exec //触发EXEC  
(nil) //事务无法被执行

### codis

## ftp

### ftp

### ftps

# 工具和插件

## Eclipse

## Svn

## Git

## Maven

### Maven生命周期

一个完整的项目构建过程通常包括清理、编译、测试、打包、集成测试、验证、部署等步骤，Maven从中抽取了一套完善的、易扩展的生命周期。Maven的生命周期是抽象的，其中的具体任务都交由插件来完成。Maven为大多数构建任务编写并绑定了默认的插件，如针对编译的插件：maven-compiler-plugin。用户也可自行配置或编写插件。

maven包括三套生命周期：clean、default、site，每个生命周期都包含了一些阶段（phase）。三套生命周期相互独立，但各个生命周期中的phase却是有顺序的，且后面的phase依赖于前面的phase。执行某个phase时，其前面的phase会依顺序执行，但不会触发另外两套生命周期中的任何phase。

#### Clean：清除编译生成的文件，有三个阶段

1）pre-clean:执行清除前的工作

2）clean:清理上一次构建生成的所有文件

3）post-clean:执行清除后的工作

#### Default： default生命周期是最核心的，它包含了构建项目时真正需要执行的所有步骤。

1）validate

2）initialize

3）generate-sources

4）process-sources

5）generate-resources

6）process-resources    ：复制和处理资源文件到target目录，准备打包；

7）compile    ：编译项目的源代码；

8）process-classes

9）generate-test-sources

10）process-test-sources

11）generate-test-resources

12）process-test-resources

13）test-compile    ：编译测试源代码；

14）process-test-classes

15）test    ：运行测试代码；

16）prepare-package

17）package    ：打包成jar或者war或者其他格式的分发包；

18）pre-integration-test

19）integration-test

20）post-integration-test

21）verify

22）install    ：将打好的包安装到本地仓库，供其他项目使用；

23）deploy    ：将打好的包安装到远程仓库，供其他项目使用；

#### Site：生成项目的站点文档

1、pre-site:

2、site: 生成项目的站点文档；

3、post-site:

4、site-deploy:发布生成的站点文档

### Maven插件

Maven所有工作都是由插件来完成，插件分为两类：

1、Build plugins 在构建项目的时候执行，应该被配置在 <build> 元素中

2、Reporting plugins 在生成站点的时候执行，应该被配置在 <reporting> 元素中

所有的插件配置要求有三个信息：groupId ， artifactId ，version ，这跟依赖的配置相似。类似 <dependencyManagement> ， 插件配置也有 <pluginManagement> ，用法也一样，参考依赖管理。

一个普通的配置看起来如下：

|  |
| --- |
| <project>  ...  <build>  <plugins>  <plugin>  <artifactId>maven-myquery-plugin</artifactId>  <version>1.0</version>  <configuration>  <url>http://www.foobar.com/query</url>  <timeout>10</timeout>  <options>  <option>one</option>  <option>two</option>  <option>three</option>  </options>  </configuration>  </plugin>  </plugins>  </build>  ...  </project> |

<configuration> 里面的元素对应插件目标的参数，如果想知道某个插件目标的可用参数，通常可以通过下面的命令查询

mvn <pluginName>:help -Ddetail -Dgoal=<goalName>

比如想知道 install 插件的 install 目标的参数，则可执行命令

mvn install:help -Ddetail -Dgoal=install

**插件目标的配置**

通常我们需要配置插件目标执行时的参数，看下面的例子

|  |
| --- |
| <project>  ...  <build>  <plugins>  <plugin>  <artifactId>maven-myquery-plugin</artifactId>  <version>1.0</version>  <executions>  <execution>  <id>execution1</id>  <phase>test</phase>  <configuration>  <url>http://www.foo.com/query</url>  <timeout>10</timeout>  <options>  <option>one</option>  <option>two</option>  <option>three</option>  </options>  </configuration>  <goals>  <goal>query</goal>  </goals>  </execution>  <execution>  <id>execution2</id>  <configuration>  <url>http://www.bar.com/query</url>  <timeout>15</timeout>  <options>  <option>four</option>  <option>five</option>  <option>six</option>  </options>  </configuration>  <goals>  <goal>query</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build>  ...  </project> |

即将 <configuration> 置于 <execution> 标签中，<execution> 标签中通过配置 <phase> 、<goal> 分别指定了配置应用的阶段和目标，如例子中的 id 为 execution1 的配置会应用在 test 阶段中的 query 目标中。我们可以看到 id 为 execution2 的 <execution> 中没有 phase 标签，那么它会在什么时候应用呢？

如果该目标默认绑定了一个阶段，则在这个阶段应用。

如果该目标没有默认的绑定，则不会应用。

这里的 <id>execution1</id> 有什么用呢？其实当我们执行一条命令时，像

mvn maven-myquery-plugin:query

这时它会应用什么配置呢？如果在 <executions> 外有配置，则会应用，如果没有，则上面配置的 <execution> 并不会应用上，那么如果我们希望执行上面配置好参数的目标，那么可以加上 id 执行，如

mvn maven-myquery-plugin:query@execution1

执行时就会应用上 execution1 的配置。

**忽略继承**

默认情况下，子工程会继承父工程的插件配置，如果不希望继承，则可配置 <inherited> 标签

<project>

...

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-antrun-plugin</artifactId>

<version>1.2</version>

<inherited>false</inherited>

...

</plugin>

</plugins>

</build>

...

</project>

### Maven常用命令

1、mvn clean:清理项目生产的临时文件,一般是模块下的target目录

2、mvn package:项目打包工具,会在模块下的target目录生成jar或war等文件，如下运行结果

3、mvn test:测试命令,或执行src/test/java/下junit的测试用例.

4、mvn install:模块安装命令 将打包的的jar/war文件复制到你的本地仓库中,供其他模块使用

5、mvn deploy:发布命令 将打包的文件发布到远程参考,提供其他人员进行下载依赖 ,一般是发布到公司的私服

6、mvn compile:编译项目源代码

7、mvn test-compile:编译测试源代码

### Pom.xml文件详解

实际上pom之间存在三种关系：继承、依赖、聚合。

#### 基本信息

|  |  |
| --- | --- |
| modelVersion | Maven模块版本，目前我们一般都取值4.0.0 |
| groupId | 整个系统的名称。 |
| artifactId | 子模块名称。 |
| packaging | 打包类型，可取值：jar,war等等，这个配置用于package的phase，具体可以参见package运行的时候启动的plugin，后面有机会我们会讲述如何配置打包的插件。 |

#### Dependence：依赖

|  |  |
| --- | --- |
| groupId | 依赖项的groupId |
| artifactId | 依赖项的artifactId |
| version | 依赖项的版本 |
| scope | 依赖项的适用范围：   * compile，缺省值，适用于所有阶段，会随着项目一起发布。 * provided，类似compile，期望JDK、容器或使用者会提供这个依赖。如servlet.jar。 * runtime，只在运行时使用，如JDBC驱动，适用运行和测试阶段。 * test，只在测试时使用，用于编译和运行测试代码。不会随项目发布。 * system，类似provided，需要显式提供包含依赖的jar，Maven不会在Repository中查找它。   之前例子里的junit就只用在了test中。 |
| exclusions | 排除项目中的依赖冲突时使用。 |
| type | 如果该依赖不是 jar 类型，则需要指定 type ，默认是 jar。如果是 war ，则需指定 <type>war<type> |

#### Parent：继承

<parent>

<groupId>com.thoughtworks.xstream</groupId>

<artifactId>xstream-parent</artifactId>

<version>1.4.3</version>

</parent>

#### Module：聚合

<modules>

<module>module1</module>

<module>module2</module>

</modules>

#### Properties：属性

定义一些常量

#### dependencyManagement：依赖管理

依赖管理只是管理依赖，并没有真正引入，一般在父pom中管理依赖，在子pom引入依赖。

<dependencyManagement>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

</dependencyManagement>

#### PluginManagement：插件管理

插件管理只是管理插件，并没有真正引入。用法跟dependencyManagement类似

# 服务器

## Tomcat

## Jetty

## Nginx

# Linux

## 常见命令

## 日志查找

## 文件查找

# 页面前端框架

## Bootstrap3

### 全局CSS样式

#### HTML5文档类型

Bootstrap 使用到的某些 HTML 元素和 CSS 属性需要将页面设置为 HTML5 文档类型。在你项目中的每个页面都要参照下面的格式进行设置。

比如：<nav>标签

|  |
| --- |
| <!DOCTYPE html>  <html lang="zh-CN">  ...  </html> |

#### 移动设备优先

在 Bootstrap 2 中，我们对框架中的某些关键部分增加了对移动设备友好的样式。而在 Bootstrap 3 中，我们重写了整个框架，使其一开始就是对移动设备友好的。这次不是简单的增加一些可选的针对移动设备的样式，而是直接融合进了框架的内核中。也就是说，**Bootstrap 是移动设备优先的**。针对移动设备的样式融合进了框架的每个角落，而不是增加一个额外的文件。

为了确保适当的绘制和触屏缩放，需要在 <head> 之中**添加 viewport 元数据标签**。

|  |
| --- |
| <meta name="viewport" content="width=device-width, initial-scale=1"> |

在移动设备浏览器上，通过为视口（viewport）设置 meta 属性为 user-scalable=no 可以禁用其缩放（zooming）功能。这样禁用缩放功能后，用户只能滚动屏幕，就能让你的网站看上去更像原生应用的感觉。注意，这种方式我们并不推荐所有网站使用，还是要看你自己的情况而定！

|  |
| --- |
| <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no"> |

#### 排版和连接

Bootstrap 排版、链接样式设置了基本的全局样式。分别是：

* 为 body 元素设置 background-color: #fff;
* 使用 @font-family-base、@font-size-base 和 @line-height-base 变量作为排版的基本参数
* 为所有链接设置了基本颜色 @link-color ，并且当链接处于 :hover 状态时才添加下划线

这些样式都能在 scaffolding.less 文件中找到对应的源码。

#### 布局和容器

Bootstrap 需要为页面内容和栅格系统包裹一个 .container 容器。我们提供了两个作此用处的类。注意，由于 padding 等属性的原因，这两种 容器类不能互相嵌套。

.container 类用于固定宽度并支持响应式布局的容器。

Copy

<div class="container"> ... </div>

.container-fluid 类用于 100% 宽度，占据全部视口（viewport）的容器。

Copy

<div class="container-fluid"> ... </div>

#### 栅格系统

Bootstrap 提供了一套响应式、移动设备优先的流式栅格系统，随着屏幕或视口（viewport）尺寸的增加，系统会自动分为最多12列。

栅格系统用于通过一系列的行（row）与列（column）的组合来创建页面布局，你的内容就可以放入这些创建好的布局中。下面就介绍一下 Bootstrap 栅格系统的工作原理：

* “行（row）”必须包含在 .container （固定宽度）或 .container-fluid （100% 宽度）中，以便为其赋予合适的排列（aligment）和内补（padding）。
* 通过“行（row）”在水平方向创建一组“列（column）”。
* 你的内容应当放置于“列（column）”内，并且，只有“列（column）”可以作为行（row）”的直接子元素。
* 类似 .row 和 .col-xs-4 这种预定义的类，可以用来快速创建栅格布局。Bootstrap 源码中定义的 mixin 也可以用来创建语义化的布局。
* 通过为“列（column）”设置 padding 属性，从而创建列与列之间的间隔（gutter）。通过为 .row 元素设置负值 margin 从而抵消掉为 .container 元素设置的 padding，也就间接为“行（row）”所包含的“列（column）”抵消掉了padding。
* 负值的 margin就是下面的示例为什么是向外突出的原因。在栅格列中的内容排成一行。
* 栅格系统中的列是通过指定1到12的值来表示其跨越的范围。例如，三个等宽的列可以使用三个 .col-xs-4 来创建。
* 如果一“行（row）”中包含了的“列（column）”大于 12，多余的“列（column）”所在的元素将被作为一个整体另起一行排列。
* 栅格类适用于与屏幕宽度大于或等于分界点大小的设备 ， 并且针对小屏幕设备覆盖栅格类。 因此，在元素上应用任何 .col-md-\*栅格类适用于与屏幕宽度大于或等于分界点大小的设备 ， 并且针对小屏幕设备覆盖栅格类。 因此，在元素上应用任何 .col-lg-\*不存在， 也影响大屏幕设备。

通过研究后面的实例，可以将这些原理应用到你的代码中。

## Jquery

## Highcharts

# Java虚拟机

# 面试技巧